LX4380 Data Sheet

Lexra, Inc.

Release 2.1

February 25, 2002

Lexra Proprietary and Confidential

LX4380 Data Sheet Revision 1.4, for RTL Release 2.1.

Table of Contents

List of Tables

List of Figures

1. Product Overview

1.1. Introduction

This data sheet describes Lexra's LX4380 processor core, a RISC processor developed for Intellectual Property (IP) licensing. The LX4380 is a carefully engineered extension to the instustry-starndard MIPS-I® ISA. The major subsystems are: the CPU core, Local Memory Interfaces (LMI) and LBus Controller (LBC). The technology includes optional interfaces to a customer-defined Coprocessors (CI[1-2]) and Custom Engine (CE) that provide extensions to the instruction set.

The LX4380 is an upgrade to the LX4189, adding a 7-stage pipeline, optional data cache support for write-back policy and 2-way set associativity, and an optional Memory Management Unit.

Features introduced in Lexra's RISC product line support System-on-Chip (SoC) design, including customer-defined Coprocessors and customer extensions to the MIPS ISA, are standard in the LX4380. Configuration options include Enhanced JTAG (EJTAG) support for debug and In-Circuit Emulation (ICE).

Because the LX4380 executes the MIPS instruction set[1], a wide variety of third-party software tools are available including compilers, operating systems, debuggers and in-circuit emulators. The assembler extensions and a cycle accurate Instruction Set Simulator (ISS) are supplied by Lexra. Programmers can use "off-the-shelf" C Compilers for initial coding; then replace performance-critical loops with optimized assembler code.

Code development tool support is provided by Lexra and by third-parties for GNU tools and by GreenHills Software for the MULTI 2000 IDE.

Key Features

- **Complete Processor Core**
  - High-performance 7-stage pipeline.
  - Executes MIPS-I ISA.
  - Extensive third-party tool support.
- **System Level Building Blocks**
  - Optional R3000-style MMU or Simplified MMMU (SMMU)
  - Optional Multiply Accumulate Unit (MAC).
  - Local instruction and/or cache interfaces, configurable sizes.
  - Local data memory and/or cache interfaces, configurable sizes.
  - Optional customer-defined coprocessors.
  - Optional customer-defined instruction extensions.
  - System bus controller.
  - Optional EJTAG Draft 2.0.0 support for debugging.

1. Except unaligned loads stores, which are executed as NOPs.

- **Portable RTL Model**
  - Available as a synthesizable RTL.
  - Portable to any 0.18µm**,** 0.15µm or 0.13µm process.
  - Support for any third-party logic and SRAM libraries.
  - Foundry partners include TSMC and UMC.
- **Easy ASIC Integration**
  - Exclusive use of positive-edge clocking.
  - Fully synchronous design.
  - System Level Building Blocks provide easy ASIC interfaces.
  - Supports for popular EDA tools.
  - User-configurable local memory, reset method, clock distribution.
  - User-configurable EJTAG breakpoints.
  - Over 30 other configuration options.

1.2. LX4380 Processor Overview

The LX4380 is a RISC processor that executes the MIPS-I instruction set[1] along with Lexra's extensions. The clocking, pipeline structure, pin-out, and memory interfaces have all been developed by Lexra to reflect system-on-silicon design needs, deep sub-micron process technology, as well as design methodology advances.

Figure 1 shows the structure of the LX4380 processor.

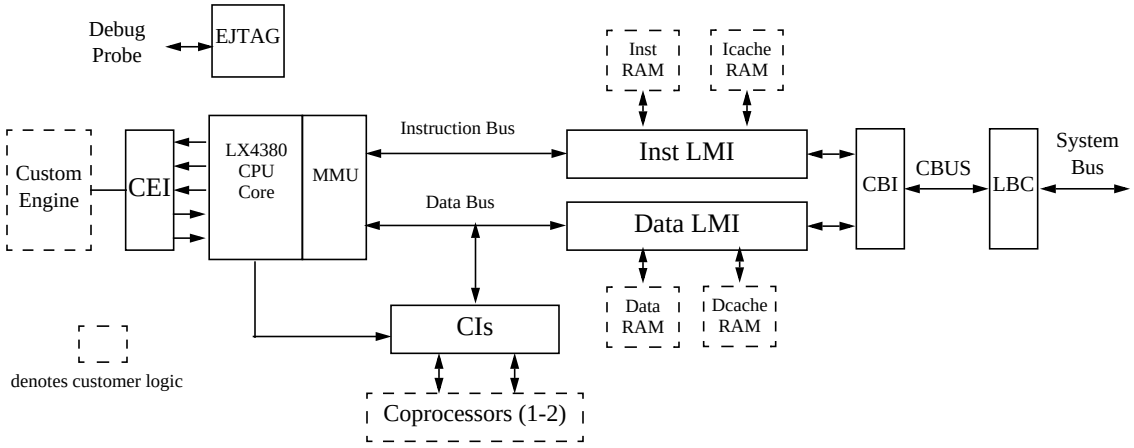


Figure 1: LX4380 Processor Overview

MIPS I Execution. The LX4380 supports the MIPS-I programming model. Two source operands can be supplied and one destination update performed per cycle. The second operand is either a register or 16-bit immediate. The instruction set includes a wide selection of ALU operations executed by the RALU, Lexra's proprietary register based ALU. The RALU also generates memory addresses for 8-bit, 16-bit and 32-bit register loads from (stores to) memory by adding a register base to an immediate offset. Branches are based on comparisons between registers, rather than flags, and are therefore easy to relocate. Optional links following jump or branch instructions assist with subroutine programming.

1. The MIPS unaligned load and store instructions (LWL, LWR, SWL, SWR) are executed as NOPs..

Pipeline. LX4380 instructions are executed by a seven-stage pipeline that has been designed so that all transactions internal to the LX4380, as well as at the interfaces, occur on the positive edge of the processor clock. Two-phase clocks are not used. The seven-stage pipeline allocates a full address-register-to-data-output-register clock cycle to both local instruction access and data access. As a result, the memories have the best timing specification possible and are decoupled from critical paths internal to the processor.

Exception Handling. The MIPS R3000 exception model is supported. Exceptions include both instruction-synchronous *traps* as well as hardware and software *interrupts*. The CP0 STATUS register controls the interrupt mask and operating mode. Exceptions are prioritized. When an exception is taken, control is transferred to the exception vector, the current instruction address is saved in the EPC register, and the exception source is identified in the CP0 CAUSE register. In the event of an address error exception, the CP0 BADVADDR register holds the failing address. For the MIPS exceptions, a program located at the exception vector identifies the cause of the exception, and transfers control to the application-specific handler. In addition to the MIPS R3000 exceptions, the LX4380 supports up to eight prioritized, vectored interrupts to meet hard real-time response requirements.

Coprocessor Instructions. The LX4380 supports the MIPS-I Coprocessor instructions These include moves to and from the 32-bit Coprocessor general registers and control registers (MTCz, MFCz, CTCz, CFCz), 32-bit Coprocessor loads and stores (LWCz, SWCz) and branches based on Coprocessor condition flags (BCzT, BCzF).

Performance and Ease of Use. The LX4380 provides excellent price/performance and time-to-market. There are two strategies used to achieve this:

- Deliver simple building blocks outside the processor core to enable system level customizations such as coprocessors, application specific instructions, memories, and busses.
- Deliver either a fully synthesizable Verilog source model or fully implemented hard core (called SmoothCore™) for customer-selected foundries.

Section 1.3 describes the System Level Building Blocks, and Section 1.4 describes the licensing models.

1.3. System Level Building Blocks

The LX4380 processor is designed to easily fit into different target applications. It provides the following building blocks.

- A Memory Management Unit (MMU). Options as either R3000-style with Translation Lookaside Buffer (TLB) or a low-cost Simplified MMU (SMMU) for deeply embedded applications.
- An optional Multiply Accumulate unit (MAC).
- A flexible Local Memory Interface (LMI) that supports instruction cache, instruction RAM, data cache and data RAM.
- Up to two Coprocessor Interfaces (CI).
- An optimized Custom Engine Interface (CEI).
- A simplified cache bus interface (CBUS) for simplified connection to peripheral devices and main memory.
- An optional Lexra Bus Controller (LBC) and Lexra Bus (LBUS) protocol for connection

to peripheral devices and main memory.

- Optional EJTAG support for embedded debug.

The following sections discuss each of these system building block interfaces.

1.3.1. Multiply-Accumulate (MAC)

The optional MAC provides 32-bit HI and LO accumulators for 32-bit multiply and divide operations and 16-bit and 32--bit multiply-accumulate operations. The MAC is pipelined for optimal performance, but also manages resource conflicts to simplify programming and debug.

1.3.2. Memory Management Unit (MMU)

The optional LX4380 MIPS R3000-style MMU is designed to permit code to run under major operating systems such as Linux, that require a Translation Lookaside Buffer (TLB) for robust protection of third-party programs and data. The MMU includes a 3-entry Instruction Translation Lookaside Buffer (ITLB), a 3-entry Data Translation Lookaside Buffer (DTLB), and a 16, 32 or 64-entry (RTL-configurable) joint Translation Lookaside Buffer (TLB).

Alternatively, for deeply embedded applications using a single address space, a Simplified Memory Management Unit (SMMU) is available. The primary function of the SMMU is to provide memory protection between user space and kernel space. The SMMU is consistent with the MIPS address space scheme for User/Kernel modes, mapping, and cached/uncached regions.

1.3.3. Local Memory Interface (LMI)

The LX4380's Harvard Architecture provides Local Memory Interfaces (LMIs) that support instruction memory and data memory. Synchronous memory interfaces are employed for all memory blocks. The LMI block is designed to easily interface with standard memory blocks provided by ASIC vendors or by third-party library vendors.

The LMIs provide direct-mapped or two-way set associative instruction cache interface, and direct-mapped or two-way set associative data cache interface. The data cache can be selected to be either write-through or write-back. The tag compare logic as well as a cache replacement algorithm are provided as part of the LMI. One of the instruction cache sets may be locked down as un-swappable local memory. Lexra's seven-stage execution pipeline provides output registers in both the instruction and data LMIs so that the memories have the best timing specification possible and are decoupled from critical paths internal to the processor.

1.3.4. Coprocessor Interface (CI)

Lexra supplies an optional Coprocessor Interface (CI) for applications that use a custom coprocessor. Up to two CIs may be employed in one design. The Coprocessor Interface "eavesdrops" on the instruction bus. If a coprocessor load (LWCz) or "move to" (MTCz, CTCz) instruction is decoded, data is passed over the data bus into a CI register, then supplied to the customer-designed coprocessor. Similarly, if a coprocessor store (SWCz) or "move from" (MFCz, CFCz) instruction is decoded, data is obtained from the coprocessor and loaded into a CI register, then transferred onto the data bus in the following cycle. The CI includes a data bus, five-bit address, and independent read and write selects for coprocessor general registers and control registers. The LX4380 pipeline and Harvard Architecture permit single cycle coprocessor access and transfer. An application-defined coprocessor condition flag is synchronized by the CI then passed to the LX4380 sequencer for testing in branch instructions.

1.3.5. Custom Engine Interface (CEI)

The LX4380 includes a Custom Engine Interface (CEI) that the application may use to extend the MIPS I ALU opcodes with application-specific or proprietary operations. Similar to the standard ALU, the CEI supplies the Custom Engine two input 32-bit operands, SRC1 and SRC2. One operand is selected from the Register File. Depending on the most significant 6 bits of the opcode, the second operand is either selected from the Register File or is a 16-bit sign-extended immediate. The opcode is locally decoded by the custom engine, and following execution by the custom engine, the result is returned on the 32-bit result bus to the LX4380. To support multi-cycle operations, a stall input is included in the interface.

1.3.6. Cache Bus (CBUS) Interface

The CBUS interface is a simple signalling layer between the LX4380 processor's cache controllers and the optional LX4380 system bus interface, the LBC. LX4380 applications that do not require the full feature set of the LBC, or that connect to a bus protocol other than LBUS, may optionally eliminate the LBC and provide their own system bus interfaces or devices that connect directly to the LX4380 using the CBUS interface. The LX4380's CBUS interface includes a built-in write buffer of configurable depth to optimize the performance of writes that occur as a result write-through data cache operation, or writes that miss the write-back data cache.

1.3.7. Lexra Bus Controller (LBC)

The optional Lexra Bus Controller (LBC) is the interface between the LX4380 and system bus devices, which may include DRAM and various peripherals. The LBC implements Lexra's LBUS protocol, a non-multiplexed, non-pipelined bus to provide a simple bus protocol for design integration. On the processor side, the LBC connects to the LX4380 CBUS. On the system side, the LBC is designed to easily interface to industry standard bus protocols, such as PCI, USB, and FireWire. The LBC supports synchronous modes with the LBUS operating at full CPU speed or half CPU speed, and an asynchronous mode that allows the LBUS to be clocked at any speed independent of the CPU speed.

1.3.8. EJTAG Debug Support

The LX4380 provides optional EJTAG (Enhanced JTAG) debug support. EJTAG allows third party hardware probes and debug software to access the processor and its attached devices in the same way the processor would access those devices. EJTAG also supports single-step instruction execution, instruction breakpoints and data breakpoints.

1.3.9. Building Block Integration

The LX4380 configuration script, *lconfig*, provides a menu of selections for designers to specify building blocks needed, number of different memory blocks, target speed, and target standard cell library. Next, the configuration software automatically generates a top level Verilog model, makefiles, and scripts for all steps of the design flow.

For testability purposes, all building blocks contain scan control signals. The Lexra synthesis scripts support optional scan insertion, which allows ATPG testing of the entire LX4380 core.

1.4. RTL Core & SmoothCore Licensing Models

Lexra delivers LX4380 as either an RTL Model or SmoothCore.

RTL Model: For standard ASIC designs, the RTL Model is fully synthesizable and scan-testable Verilog source code, and may be targeted to any ASIC vendor's standard cell libraries. In this case, the designer may

simply follow the ASIC vendor's design flow to ensure proper sign-off. In addition to the Verilog source code and system level test bench, Lexra provides synthesis scripts as well as floor plan guidelines to maximize the performance of the LX4380.

SmoothCore: For COT designs that are manufactured at foundries such as TSMC and UMC, a SmoothCore port is the quickest, lowest cost, and best performance choice. Lexra provides a porting service that delivers a fully implemented and verified hard macro for a customer-specific configuration, foundry and library. All data path, register file, and interface optimizations are performed by Lexra to ensure the smallest die size and fastest performance possible. A scan based test pattern is provided for fault coverage during manufacturing tests.

1.5. EDA Tool Support

Lexra supports mainstream EDA software, so designers do not have to alter their design methodology. The following is a snapshot of EDA tools currently supported:

Table 1: EDA Tool Support

Design Flow	Tools Supported
Simulation	Synopsys VCS Cadence Verilog XL Cadence NC-Verilog
Synthesis	Synopsys Design Compiler
Static Timing	Synopsys PrimeTime
DFT	Synopsys TetraMax
P&R	Avant! Apollo II

2. Architecture

2.1. Hardware Architecture

The LX4380 processor includes the Control Processor (CP0) and the register file and ALU (RALU), as well as the optional Memory Management Unit (MMU). CP0 includes instruction address sequencing and exception processing. The RALU performs ALU operations and generates data addresses. The MMU converts the Virtual Page Numbers (VPNs) in the instruction address and data address into Page Frame Numbers (PFNs).

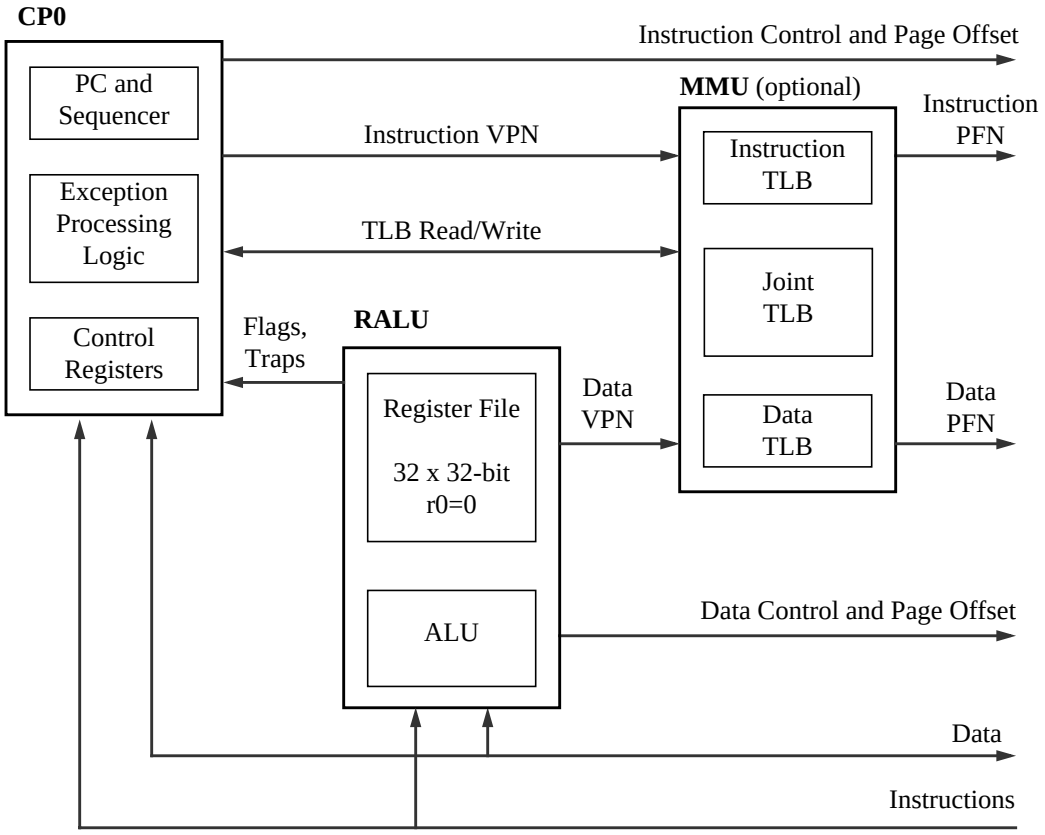


Figure 2: Processor Core Module Partitioning

2.2. Seven Stage Pipeline

The LX4380 has a seven stage pipeline:

stage	*name*	*actions*
1	I	Instruction fetch
2	D	Decode instruction
3	S	Source operand fetch (register file read)
4	E	Execute ALU operations memory address generation
5	A	Access data memory (read data cache store and tags)
6	M	Memory data select and format
7	W	Write data to register file and memory

The seven stage pipeline provides a complete processor cycle for the instruction memory and data memory accesses, allowing use of larger memories and 2-way set-associative caches without degrading cycle time. The seven pipeline stages allow the processor clock speed to scale with current silicon processes.

A two cycle penalty is incurred on branch prediction failure. However, the LX4380's conditional move instructions can be used to avoid any wasted cycles in the control of real-time critical loops.

2.3. RALU Data Path

The LX4380 RALU incorporates a 32x32b four-port register file. One write port is dedicated to 32-bit register file loads from the Data Bus (Loads, MFCz, CFCz - moves from coprocessor). The remaining three ports (2r/1w) are used for the other operations, such as ALU operations.

The instruction set includes a wide selection of ALU operations executed by the RALU. In the case of ALU operations, one operand is a register and the second operand is either a register or 16-bit immediate value. The immediate value is sign-extended or zero-extended, depending on the operation. Signed adds and subtracts can generate the arithmetic overflow trap, Ov, which is sampled by CP0.

The RALU also generates the virtual memory addresses for register loads from (stores to) memory by adding a register base to a sign-extended 16-bit immediate offset. Data address errors generate the *AdEL, AdES* trap flags which are sampled by CP0. The LX4380 employs *Big-Endian* memory addressing.

Branches are based on comparisons between registers, rather than implicit flags, permitting the programmer more flexibility. From these comparisons, the RALU generates *N* and *Z* flags for sampling in CP0. Branch or jump instructions may optionally store in a general purpose register the address of the instruction at the memory location following the branch delay slot of a jump or a branch which is taken. This register, called the *link*, holds the return address following a subroutine call.

Coprocessor operations permit moves of the general purpose registers to or from optional application-specific coprocessors (one or two). These transfers occur over the Data Bus, similar to data memory loads and stores.

2.4. System Control Coprocessor (CP0)

The System Control Coprocessor (CP0) is responsible for instruction address sequencing and exception processing.

For normal execution, the next instruction address has several potential sources: the increment of the previous address, a branch address computed using a pc-relative offset, or a jump target address. For jump addresses, the absolute target can be included in the instruction, or it can be the contents of a general-purpose register transferred from the RALU.

Branches are assumed (or predicted) to be taken. In the event of prediction failure, two stall cycles are

incurred and the correct address is selected from a special “backup” register. Statistics from several large programs suggest that these stalls will degrade average LX4380 throughput by several percent. However, the net effect of the LX4380’s branch prediction on performance is positive because this technique eliminates certain critical paths and therefore, permits a higher speed system clock.

If an *exception* occurs, CP0 selects one of several hardwired vectors for the next instruction address. The exception vector depends on the mode and specific trap which occurred. This is described further in Section 3.4, Exception Processing.

The following registers, which are visible to the programming model, are located in CP0:

Table 2: CP0 Registers

CP0 register	Number	Function
INDEX	0	TLB location for direct read or write, or result of TLB probe
RANDOM	1	A pseudo-random value specifying a TLB location to write
ENTRYLO	2	Low part of TLB entry written to or read from TLB
CONTEXT	4	Accelerates page table lookup in service of TLB miss
WIRED	6	Indicates number of wired TLB entries
BADVADDR	8	Holds bad virtual address if address exception error occurs
ENTRYHI	9	High part of TLB entry written to or read from TLB
STATUS	12	Interrupt masks, mode selects
CAUSE	13	Exception cause
EPC	14	Holds address for return after exception handler
PRID	15	Processor ID (read-only) 0x0000cd01 for LX4380
DREG	16	EJTAG debug control
DEPC	17	EJTAG debug exception PC
CCTL	20	Instruction and data memory control
DESAVE	31	EJTAG debug save register

EPC, STATUS, CAUSE, and BADVADDR are described in the Section 3.4. The INDEX, RANDOM, ENTRYLO, CONTEXT and ENTRYHI registers are described in Section 5. The DREG, DEPC and DESAVE registers are used by EJTAG probe debug software, and are described in the EJTAG 2.0.0 specification. The PRID register is a read-only register that allows software to identify the Lexra processor model. The CCTL register is a Lexra defined CP0 register used to control the instruction and data memories, as described in Section 7.2.

The contents of the registers listed in Table 2 are transferred to and from the RALU’s general-purpose register file by the MFC0 and MTC0 instructions, as described in Section 3.1.6.

The LX4380 implements extended CP0 registers that provide additional functions summarized in Table 3.

Table 3: Extended CP0 Registers

CP0 register	Number	Function
ESTATUS	0	Interrupt masks for prioritized vectored interrupts.
ECAUSE	1	Interrupt pending flags for prioritized vectored interrupts.
INTVEC	2	Address of vector table for prioritized vectored interrupts.

The registers listed in Table 3 are described in detail in Section 3.5. The contents of these registers are transferred to an from the RALU's general-purpose register file by the MFLXC0 and MTLXC0 instructions, as described in Section 3.1.6.

3. RISC Programming Model

This section describes the LX4380 programming model. Section 3.1 summarizes the basic RISC operations supported by the LX4380. These opcodes may be extended by the customer using Lexra's Custom Engine Interface (CEI). This capability is described in Section 3.2.

Section 3.3 describes the Simple Memory Management Unit (SMMU). The SMMU provides sufficient memory management capabilities for most embedded applications while supporting execution of third-party MIPS software development tools. (Section 5 describes the optional programmable MMU.)

The LX4380 supports the MIPS R3000 exception processing model, as described in Section 3.4.

The LX4380 supports MIPS I coprocessor operations. The customer can include one or two application-specific coprocessors. The LX4380 includes an optional Coprocessor Interface (CI) that provides a simplified connection between a coprocessor and the internal signals of the LX4380. The CI is described in Section 3.6.

3.1. Summary of Basic RISC Instructions

The LX4380 executes the MIPS I (R2000/R3000) instructions as detailed in the tables below. The LX4380 executes full MIPS I instruction set, excluding the unaligned load and store instructions (LWL, SWL, LWR, SWR) which are executed as no-ops.

The MULT, MULTU, DIV and DIVU are MIPS I instructions that are supported by the LX4380's optional MAC module. These instructions are described in section Section 4, Integer Multiply-Accumulate.

The MTLXC0, MFLXC0, MOVZ, MOVN and LTW instructions shown in this section are not MIPS I instructions.

Additional instructions supported by the LX4380 are described in Section 4.2, MAC Instructions; Section 5.3, TLB Instructions; and Section 7.3, CACHE Instruction.

The following conventions are employed in the instruction descriptions.

« »	Encloses a list of syntax choices, from which one must be chosen.
{ }	Encloses a list of values that are concatented to form a larger value.
n { value }	Replicates (concatenates) *value* n times.
value[3]	Bits selected from *value*.
[rS + offset]	Memory address computation and corresponding memory contents.
4'b0000	A sized constant binary value.
32'h1234_5678	A sized constant hexadecimal value.
expr ? A : B	Select A if *expr* is true, otherwise select B.

3.1.1. ALU Instructions

Table 4: ALU Instructions

Instruction		Name and Description
ADD ADDU ADDI ADDIU	rD, rS, rT rD, rS, rT rD, rS, immediate rD, rS, immediate	Add Add Unsigned Add Immediate Add Immediate Unsigned rD ← rS + «rT, immediate» Add reg rS to either reg rT or a 16-bit immediate sign-extended to 32 bits. Result is stored in reg rD. ADD and ADDI can generate overflow trap; ADDU and ADDIU do not.
SUB SUBU	rD, rS, rT rD, rS, rT	Subtract Subtract Unsigned rD ← rS - rT Subtract reg rT from reg rS. Result is stored in register rD. SUB can generate overflow trap. SUBU does not.
AND ANDI	rD, rS, rT rD, rS, immediate	And And Immediate rD ← rS & «rT, immediate» Logical *and* of reg rS with either reg rT or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
OR ORI	rD, rS, rT rD, rS, immediate	Or Or Immediate rD ← rS | «rT, immediate» Logical *or* of reg rS with either reg rT or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
XOR XORI	rD, rS, rT rD, rS, immediate	Exclusive Or Exclusive Or Immediate D ← rS ^ «rT, immediate» Logical *xor* of reg rS with either reg rT or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
NOR	rD, rS, rT	Nor rD ← ~(rS | rT) Logical *nor* of reg rS with reg rT. Result is stored in reg rD.

Instruction		Name and Description
LUI	rD, immediate	Load Upper Immediate rD ← {immediate, 16'b0} The 16-bit immediate is stored into the upper half of reg rD. The lower half is loaded with zeroes.
SLL SLLV	rD, rT, immediate rD, rT, rS	Shift Left Logical Shift Left Logical Variable rD ← rT << «rS, immediate» The 5-bit shift amount *amt* is obtained from the immediate field (SLL) or bits 4:0 of reg rS (SLLV). The contents of reg rT are shifted left *amt* bits. The result is stored in reg rD.
SRL SRLV	rD, rT, immediate rD, rT, rS	Shift Right Logical Shift Right Logical (Variable) rD ← rT >> «rS, immediate» The 5-bit shift amount *amt* is obtained from the immediate field (SRL) or bits 4:0 of reg rS (SRLV). The contents of reg rT are shifted right *amt* bits. The result is stored in reg rD.
SRA SRAV	rD, rT, immediate rD, rT, rS	Shift Right Arithmetic Shift Right Arithmetic Variable rD ← rT >>(a) «rS, immediate» The 5-bit shift amount *amt* is obtained from the immediate field (SRA) or bits 4:0 of reg rS (SRAV). The contents of reg rT are arithmetic shifted right *amt* bits. The result is stored in reg rD.
SLT SLTU SLTI SLTIU	rD, rS, rT rD, rS, rT rD, rS, immediate rD, rS, immediate	Set on Less Than Set on Less Than Unsigned Set on Less Than Immediate Set on Less Than Immediate Unsigned rD ← (rS < «rT, immediate») ? 1 : 0 If reg rS is less than «rT, immediate» set rD to 1, else 0. The 16-bit immediate is sign extended. For SLT, SLTI, the comparison is signed; for SLU, SLTIU, the comparison is unsigned.

3.1.2. Load and Store Instructions

Table 5: Load and Store Instructions

Instruction		Description
LB LBU LH LHU LW	rT, offset(rS) rT, offset(rS) rT, offset(rS) rT, offset(rS) rT, offset(rS)	Load Byte Load Byte Unsigned Load Halfword Load Halfword Unsigned Load Word rT ← Memory[rS + offset] Reg rT is loaded from data memory. The memory address is computed as *base + offset*, where the base is reg rS and the offset is the 16-bit offset sign-extended to 32 bits. LB, LBU addresses are interpreted as byte addresses to data memory; LH, LHU as halfword (16-bit) addresses; LW as word (32-bit) addresses. The data fetched in LB, LH (LBU, LHU) is sign-extended (zero-extended) to 32-bits for storage to reg rT. rT cannot be referenced in the instruction following a load instruction.
LTW	rT, offset(rS)	Load TwinWord { rT, rT+1 } ← Memory[rS + offset] The *offset*, in bytes, is a signed rT 13-bit quantity that must be divisible by 8 (since it occupies only 10 bits of the instruction word). The *offset* is sign extended and added to the contents of the register rT to form the address *temp*. The word addressed by *temp* is fetched and loaded into rT (which must be an even register). The word addressed by *temp*+4 is loaded into rT+1. If *temp* is not twinword aligned, an address exception is taken. If the instruction immediately following LTW attempts to use rT or rT+1, the results of that instruction are unpredictable.
SB SH SW	rT, offset(rS) rT, offset(rS) rT, offset(rS)	Store Byte Store Halfword Store Word Memory[rS + offset] ← rT Reg rT is stored to data memory. The memory address is computed as *base + offset*, where the base is reg rS and the offset is the 16-bit offset sign-extended to 32 bits. SB addresses are interpreted as byte addresses to data memory; the 8 low-order bits of rT are stored. SH addresses are interpreted as halfword addresses to data memory; the 16 low order bits of rT are stored.

3.1.3. Conditional Move Instructions

Table 6: Conditional Move Instructions

Instruction	Description
MOVZ rD, rS, rT	Move if Zero rD ← (rT== 0) ? rS : rD If the contents of general register rT are equal to 0, the general register rD is updated with rS; otherwise rD is unchanged.
MOVN rD, rS, rT	Move if Not Zero rD ← (rT != 0) ? rS : rD If the contents of general register rT are not equal to 0, the general register rD is updated with rS; otherwise rD is unchanged.

3.1.4. Branch and Jump Instructions

Table 7: Branch and Jump Instructions

Instruction	Description
BEQ rS, rT, offset BNE rS, rT, offset	Branch if Equal Branch if Not Equal if COND pc ← pc + 4 + { 14 { offset[15] }, offset, 2'b00 } else pc ← pc + 8 where COND = (rS = rT) for EQ, (rS ne rT) for NE, and offset is a 16-bit value. For BEQ, BNE the instruction after the branch (*delay slot*) is always executed.
BLEZ rS, offset BGTZ rS, offset	Branch if Less Than or Equal to Zero Branch if Greater Than Zero if COND pc ← pc + 4 + { 14 { offset[15] }, offset, 2'b00 } else pc ← pc + 8 where COND = (rS <= 0) for LE, (rS > 0) for GT, and offset is a 16-bit value For BLEZ, BGTZ the instruction after the branch (*delay slot*) is always executed.
BLTZ rS, offset BGEZ rS, offset	Branch if Less Than Zero Branch if Greater Than or Equal to Zero if COND pc ← pc + 4 + { 14 { offset[15] }, offset, 2'b00 } else pc ← pc + 8 where COND = (rS < 0) for LT, (rS >= 0) for GE, and offset is a 16-bit value For BLTZ, BGEZ the instruction after the branch (*delay slot*) is always executed.
BLTZAL rS, offset BGEZAL rS, offset	Branch if Less Than Zero And Link Branch if Greater Than or Equal to Zero And Link Similar to the BLTZ and BGEZ except that the address of the instruction following the delay slot is saved in r31 (regardless of whether the branch is taken.)

Instruction		Description
J	target	Jump pc ← { pc[31:28], target, 2'b00 } The jump target is a 26-bit absolute value. The instruction following J (delay slot) is always executed.
JAL	target	Jump And Link Same as Jump (J), except that the address of the instruction following the delay slot is saved in r31.
JR	rS	Jump Register pc ← (rS) Jump to the address specified in rS. The instruction following JR (delay slot) is always executed.
JALR	rS, rD	Jump And Link Register Same as Jump Register (JR), except that the address of the instruction following the delay slot is saved in rD.

3.1.5. Control Instructions

Table 8: Control Instructions

Instruction	Description
SYSCALL	System Call The Sys Trap occurs when SYSCALL is executed.
BREAK	Break The Bp Trap occurs when BREAK is executed.
RFE	Restore From Exception Causes the KU/IE stack to be popped. Used when returning from the exception handler. See "Exception Processing" below.

3.1.6. Coprocessor Instructions

Table 9: Coprocessor Instructions

Instruction	Description
LWCz rCGEN, offset(rS)	Load Word to Coprocessor Z rCGEN ← Memory[rS + offset] Coprocessor z [1-3] general reg rCGEN is loaded from data memory. The memory address is computed as *base + offset*, where the base is reg rS and the offset is the 16-bit offset sign-extended to 32 bits. rCGEN cannot be referenced in the following instruction (one cycle delay).
SWCz rCGEN, offset(rS)	Store Word from Coprocessor Z Memory[rS + offset] ← rCGEN Coprocessor z [1-3] general reg rCGEN is stored to data memory. The memory address is computed as *base + offset*, where the base is reg rS and the offset is the16-bit offset sign-extended to 32 bits.
MTCz rT, rCGEN CTCz rT, rCCON	Move To Coprocessor Z Move Control To Coprocessor Z In MTCz(CTCz), the general register rT is moved to coprocessor z [0-3] general (control) reg rCGEN(rCCON). rCGEN and rCCON cannot be referenced in the following instruction.
MFCz rT, rCGEN CFCz rT, rCCON	Move From Coprocessor Z Move Control From Coprocessor Z In MFCz (CFCz), the coprocessor z [0-3] general (control) reg rCGEN (rCCON) is moved to the general register rT. rT cannot be referenced in the following instruction.

Instruction	Description
MTLXC0 rT, LX0reg	Move To Lexra Coprocessor 0 Register The contents of general register rT are moved to the Lexra-defined CP0 register indicated by LX0reg.
MFLXC0 rT, LX0reg	Move From Lexra Coprocessor 0 Register The general register rT is loaded from the contents of the Lexra-defined CP0 register indicated by LX0reg. rT cannot be referenced in the following instruction.
BCzT offset BCzF offset	Branch if Coprocessor Z is True Branch if Coprocessor Z is False if COND pc ← pc + 4 + { 14' { offset[15] } , offset, 2'b00 } else pc ← pc + 8 where COND = (CpCondz = True) for BCzT, (CpCondz = False) for BCzF. For BCzT, BCzF the instruction after the branch (*delay slot*) is always executed.

3.2. Opcode Extension Using the Custom Engine Interface (CEI)

Customers may add proprietary or application-specific opcodes to their LX4380 based products using the Custom Engine Interface (CEI). The new instructions take one of the following forms illustrated below and use reserved opcodes.

Table 10: Custom Engine Interface Operations

New Instruction	Description	Available Opcodes
NEWOPI rD, rS, immed	New Operation Immediate rD ← rS NEWOPI immed Reg rS is supplied to the SRC1 port of CEI and the 16-bit immediate, sign-extended to 32-bits is supplied to SRC2. The result of the customer's NEWOPI is placed on the CEI input port RES and stored in reg rD.	INST[31:26] = 24 - 27
NEWOP rD, rS, rT	New Operation rD ← rS NEWOPR rT Reg rS is supplied to the SRC1 port of CEI and reg rT is supplied to SRC2. The result of the customer's NEWOPI is placed on the CEI input port RES and stored in reg rD.	INST[31:26] = 0 *and* INST[5:0] = 56,58-60,62-63

Lexra permits customer operations to be added using the four (4) I-Format opcodes and six (6) R-Format opcodes listed in the table above. Other opcode extensions in future Lexra products will *not* utilize the opcodes reserved above.

When the Custom Engine decodes NEWOPI or NEWOPR, it must signal the core that a custom operation has been executed so that the Reserved Instruction (RI) trap will not be taken. Multi-cycle custom operations may be executed by asserting the LX4380's CEI halt input.

Note: The custom operation may choose to ignore the SRC1 and SRC2 operands supplied by the CEI and reference internal Custom Engine registers instead. Results can also be written to an implicit custom register. However, unless rD = 0 is coded a register in the processor will also be written.

See the table entries under Custom Engine Interface on page 109 for a listing of the CEI signals.

3.3. Simple Memory Management Unit

The LX4380 includes a Simple Memory Management Unit (SMMU) for the instruction memory address and the data memory address. The hardwired virtual-to-physical address translation performed by the SMMU is sufficient to ensure execution of third-party software development tools.

Table 11: SMMU Address Translation

Region Name	Virtual Address	Physical Address	Cacheability	Permission
kuseg	0x0000_0000-0x7FFF_FFFF	0x4000_0000-0xBFFF_FFFF	cached	kernel or user
kseg0	0x8000_0000-0x9FFF_FFFF	0x0000_0000-0x1FFF_FFFF	cached	kernel
kseg1	0xA000_0000-0xBFFF_FFFF	0x0000_0000-0x1FFF_FFFF	uncached	kernel
kseg2	0xC000_0000-0xFEFF_FFFF	0xC000_0000-0xFEFF_FFFF	cached	kernel
upper-kseg2	0xFF00_0000-0xFFFF_FFFF	0xFF00_0000-0xFFFF_FFFF	uncached	kernel

The LX4380 includes optional support for a fully programmable MIPS R3000-style MMU. This is described in Section 5, Memory Management Unit (MMU).

3.4. Exception Processing

The LX4380 implements the MIPS R3000 exception processing model. TLB related exceptions are included only if the LX4380 is configured with the optional MMU. The term *exception* refers to *traps*, which are non-maskable program synchronous events, and *interrupts*, which result from unmasked asynchronous events.

The list below is numbered from highest to lowest priority. ExcCode is stored in CAUSE when an exception is taken. Sys, Bp, RI, CpU can share the same priority level because only one can occur in a given time slot.

Table 12: List of Exceptions

Exception	Priority	ExcCode	Description
Reset	1	--	Reset trap.
AdEL – instruction	2	4	Address exception trap. Instruction fetch. Occurs if the instruction address is not word-aligned or if a kernel address is referenced in user mode.
TLBL - instruction	3	2	TLB instruction fetch trap. Occurs when a virtual instruction address does not match a TLB entry.
Ov	4	12	Arithmetic overflow trap. Can occur as a result of signed add or subtract operations.
Sys	5	8	SYSCALL instruction trap. Occurs when SYSCALL instruction is executed.
Bp	5	9	BREAK instruction trap. Occurs when BREAK instruction is executed.
RI	5	10	Reserved instruction trap. Occurs when a reserved opcode is fetched.
CpU	5	11	Coprocessor Usability trap. Occurs when an attempt is made to execute a coprocessor z operation and coprocessor z is not enabled (via the STATUS register).
AdEL – data	6	4	Address exception trap. Data fetch. Occurs if the data address is not properly aligned or if a kernel address is generated in user mode.
AdES	7	5	Address exception trap. Data store. Occurs if the data address is not properly aligned or if a kernel address is generated in user mode.
TLBL - data	8	2	TLB data load trap. Occurs when the virtual data address of a load operation does not match a TLB entry.
TLBS	8	3	TLB data store trap. Occurs when the virtual data address of a store operation does not match a TLB entry.
TLBMOD	8	1	TLB data modified trap. Occurs when the virtual data address of a store operation matches a TLB entry that is marked valid but not dirty.
Int	9	0	Unmasked interrupt from one or more of the six R3000 non-prioritized hardware interrupt requests, or the eight Lexra-specific prioritized hardware interrupt requests.

3.4.1. Exception Processing Registers

These registers are read or written using MFC0 and MTC0 operations. The 0 fields are ignored on write and are 0 on read. To ensure compatibility with future LX4380 versions, they should be written with 0.

STATUS: Coprocessor 0 General Register Address = 12

31-28	27-23	22	21-16	15-8	7-6	5	4	3	2	1	0
CU[3:0]	0	BEV	0	IM[7:0]	0	KUo	IEo	KUp	IEp	KUc	IEc

Field	Description	R/W	Reset
CU	CU[z] = 1 (0) indicates that coprocessor z is usable (unusable) in coprocessor instructions. In kernel mode, CP0 is always usable regardless of the setting of CU[0].	R/W	0
BEV	Bootstrap Exception Vector. Selects between two trap vectors. (See Section 3.4.2.)	R/W	1
IM	Interrupt masks for the six non-prioritized hardware interrupts and two software interrupts.	R/W	0
KU/IE	KU = 0 (1) indicates kernel (user) mode. In the LX4380, user mode virtual addresses must have msb = 0. In kernel mode, the full address space is addressable. IE = 1 (0) indicates that interrupts are enabled (disabled). The KUo, IEo, KUp, IEp, KUc and IEc fields form a three-level stack hardware stack KU/IE signals. The *current* values are KUc/IEc, the *previous* values are KUp/IEp, and the *old* values (those before previous) are KUo/IEo. (See Section 3.4.2.)	R/w	0

CAUSE: Coprocessor 0 General Register Address = 13

31	30	29-28	27-16	15-8	7	6-2	1-0
BD	0	CE[1:0]	0	IP[7:0]	0	ExcCode[4:0]	0

Field	Description	R/W	Reset
BD	Branch Delay. Indicates that the exception was taken in a branch or jump delay slot.	R	0
CE	Coprocessor Exception. In the case of a Coprocessor Usability exception, indicates the number of the responsible coprocessor.	R	0
IP[7:2]	Interrupt Pending. Bits are set when the corresponding hardware interrupt input INTREQ_N[7:2] request is pending. Level sensitive.	R	0[a]
IP[1:0]	Interrupt Pending Software controllable interrupts. Level sensitive.	R/W	
ExcCode	The ExcCode values (listed in Table 12) are stored here when an exception occurs.	R	0

a. After reset is de-asserted, IP contains values sampled from hardware interrupt sources.

EPC: Coprocessor 0 General Register Address = 14

31 - 0
EPC

Field	Description	R/W	Reset
EPC	Exception Program Counter.	R/W	0

EPC contains the virtual address of the next instruction to be executed following return from the exception handler. If the exception occurs in the delay slot of a branch, EPC holds the address of the branch instruction and BD is set in Cause. The branch will typically be re-executed following the exception handler.

BADVADDR: Coprocessor 0 General Register Address = 8

31 - 0
BadVAddr

Field	Description	R/W	Reset
BadVAddr	Bad Virtual Address. Contains the virtual address (instruction or data) which generated an AdEL or AdES exception error.	R	0

3.4.2. Exception Processing: Entry and Exit

When an exception occurs, the instruction address changes to one of the following locations:

RESET	0xbfc0_0000
Other exceptions, BEV = 0	0x8000_0080
Other exceptions, BEV = 1	0xbfc0_0180

The KU/IE stack is pushed:

{ KUo, IEo, KUp, IEp, KUc, IEc } (before push)

{ KUp, IEp, KUc, IEc, 0, 0 } (after push)

which disables interrupts and puts the program in kernel mode. The code (ExcCode) for the exception source is loaded into CAUSE so that the application-specific exception handler can determine the appropriate action. The exception handler should not re-enable Interrupts until necessary information has been saved.

To return from the exception, the exception handler first moves EPC to a general register using MFC0, followed by a JR operation. RFE only *pops* the KU/IE stack:

{ KUp, IEp, KUc, IEc, 0, 0 } (before pop)

{ KUp, IEp, KUp, IEp, KUc, IEc } (after pop)

(This example assumes that KU/IE were not modified by the exception handler). Therefore, a typical sequence of operations to return from the exception handler would be:

```
MFC0        r26, C0_EPC     // r26 is a temporary storage register in the RALU
. . .
JR          r26
RFE
```

3.5. Low-Overhead Prioritized Interrupts

The LX4380 includes eight low-overhead hardware interrupt signals that extend the MIPS R3000 interrupt exception model. These signals are compatible with the R3000 exception processing model and are useful for real-time applications.

These interrupts are supported with three Lexra-defined CP0 registers, ESTATUS, ECAUSE, and INTVEC, accessed with the MTLXC0 and MFLXC0 variants of the MTC0 and MFC0 instructions. The 0 fields in these registers are ignored on write and are 0 on read. To ensure compatibility with future LX4380 versions, they should be written with 0. As with any CP0 instruction, a Coprocessor Unusable Exception is taken if these instructions are executed while in User Mode and the CU0 bit is 0 in the CP0 STATUS register.

The three Lexra CP0 registers are ESTATUS (0), ECAUSE (1), and INTVEC (2), and are defined as follows:

ESTATUS (LX CP0 Reg 0) Read/Write

31 - 24	23 - 16	15 - 0
0	IM[15:8]	0

Field	Description	R/W	Reset
IM	Interrupt masks for the eight prioritized hardware interrupts.	R/W	0

ECAUSE (LX CP0 Reg 1) Read-only

31 - 24	23 - 16	15 - 0
0	IP[15:8]	0

Field	Description	R/W	Reset
IP	Interrupt pending flags for the eight prioritized hardware interrupts.	R	0[a]

a. After reset is de-asserted, IP contains values sampled from hardware interrupt sources.

INTVEC (LX CP0 Reg 2) Read/Write

31 - 6	5 - 0
BASE	0

Field	Description	R/W	Reset
BASE	Base address of interrupt vector table (bits 31-6).	R/W	0

ESTATUS contains the interrupt mask bits IM[15:8], which are reset to 0 so that none of the vectored interrupts will be activated, regardless of the global interrupt enable flag, IEc, in the CP0 STATUS register. (See Section 3.4.1, Exception Processing Registers.) The interrupt pending flags IP[15:8] for the vectored interrupt signals are located in ECAUSE and are read-only. These fields are similar to the IM[7:0] and IP[7:0] fields defined in the R3000 exception processing model, except that the vectored interrupts are prioritized in hardware, and each has a dedicated exception vector.

IP[15] has the highest priority, while IP[8] has the lowest priority, however, all vectored interrupts are higher priority than IP[7:0]. The processor concatenates the program defined BASE address for the exception vectors with the interrupt number to form the interrupt vector, as shown in the table below. Two instructions can be executed in each vector; typically these will consist of a jump instruction and its delay slot, with the target of the jump being either a shared interrupt handler or one that is unique to that particular interrupt.

Table 13: Prioritized Interrupt Exception Vectors

Interrupt Number	Exception Vector
15	{ BASE, 6'b111000 }
14	{ BASE, 6'b110000 }
13	{ BASE, 6'b101000 }
12	{ BASE, 6'b100000 }
11	{ BASE, 6'b011000 }
10	{ BASE, 6'b010000 }
9	{ BASE, 6'b001000 }
8	{ BASE, 6'b000000 }

When a vectored interrupt causes an exception, all of the standard actions for an exception occur. These include updating the EPC register and certain sub-fields of the standard STATUS and CAUSE registers. In particular, the Exception Code of the CAUSE register indicates "Interrupt", and the "current" and "previous" mode bits of the STATUS register are updated in the usual manner.

3.6. Coprocessors

Applications may include up to two coprocessors to interface with the LX4380. The contents of these coprocessors may include up to thirty-two 32-bit *general registers* and up to thirty-two 32-bit *control registers*. The general registers may be moved to and from the RALU's registers using MTCz, MFCz operations, or be loaded and stored from data memory using LWCz, SWCz operations. The control registers may only be moved to and from the RALU's registers using CTCz, CFCz operations.

The LX4380 includes the optional Coprocessor Interface (CI) allowing the customer to easily interface a

coprocessor to the LX4380. The CI supplies a set of control, address, and data busses that may be tied directly to the coprocessor general and control registers.

The CI is described in more detail in Section 6, Coprocessor Interface.

4. Integer Multiply-Accumulate

The Multiply-Accumulate (MAC) module is an optional feature of the LX4380 processor. This section discusses the operation and features of the MAC module.

4.1. MAC Overview

- Independent 32-bit HI and LO accumulators for 16-bit multiply-accumulate improve performance by allowing the generation of a new result while a previous result is pending.
- Multiply-subtract instructions eliminate the need to negate coefficients.
- Two-bit per cycle divider writes results to HI and LO only on the last cycle.
- The hardware manages all resource conflicts to simplify software design and debug.

Figure 3 illustrates the MAC data paths. Multiply and multiply-accumulate operations proceed through a two-stage pipelined multiplier. Accumulator logic in a third pipeline stage adds the multiplication result to the HI and LO accumulators. Divide operations act on 32-bit divisor and dividend operands. Two bits of quotient and remainder are generated per cycle, and partial quotients and remainders are stored in the divider until the last cycle.

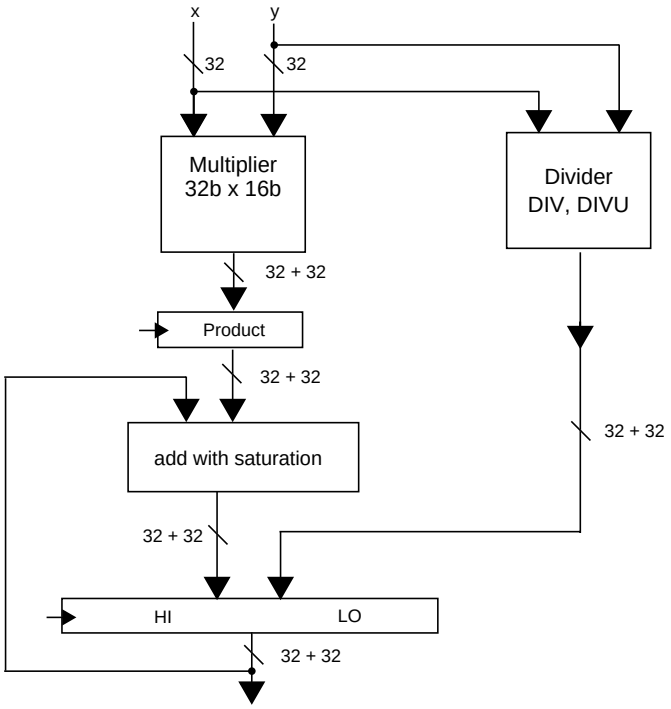


Figure 3: MAC Data Paths

4.2. MAC Instructions

Table 14: 32-Bit Multiply and Divide Instructions

MTHI rS	Move To HI HI ← rS The contents of rS are stored in the HI register.
MTLO rS	Move To LO LO ← rS The contents of rS are stored in the LO register.
MFHI rD	Move From HI rD ← HI The contents of the HI register are stored in rD.
MFLO rD	Move From LO rD ← LO The contents of the LO register are stored in rD.
MULT rS, rT	Signed 32-bit Multiply t ← rS * rT LO ← t[31:0] HI ← t[63:32] The contents of rS is multiplied by rT, treating the operands as signed 2's complement values. The upper 32-bits of the 64-bit result are stored in the HI register. The lower 32-bits are stored in the LO register.
MULTU rS, rT	Unsigned 32-bit Multiply t ← rS * rT LO ← t[31:0] HI ← t[63:32] The contents of rS is multiplied by rT, treating the operands as unsigned values. The upper 32-bits of the 64-bit result are stored in the HI register. The lower 32-bits are stored in the LO register.
DIV rS, rT	Signed Divide HI ← rS % rT LO ← rS / rT The contents of rS are divided by the contents of rT, treating the operands as signed 2's complement values. The 32-bit quotient is stored in the LO register. The 32-bit remainder is stored in the HI register.

DIVU rS, rT	Unsigned Divide HI ← rS % rT LO ← rS / rT The contents of rS are divided by the contents of rT, treating the operands as unsigned values. The 32-bit quotient is stored in the LO register. The 32-bit remainder is stored in the HI register.

Table 15: 16-Bit Multiply and Multiply-Accumulate Instructions

MAZH rS, rT	Signed 16-bit Multiply to HI HI ← 0 + rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is (added to zero and) stored in the HI register.
MAZL rS, rT	Signed 16-bit Multiply to LO LO ← 0 + rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is (added to zero and) stored in the LO register.
MADH rS, rT	Signed 16-bit Multiply-Accumulate to HI HI ← HI + rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is added to HI, ignoring any overflow. The result is stored in the HI register.
MADL rS, rT	Signed 16-bit Multiply-Accumulate to LO LO ← LO + rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is added to LO, ignoring any overflow. The result is stored in the LO register.
MSZH rS, rT	Signed 16-bit Multiply-Negate to HI HI ← 0 - rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is negated (subtracted from zero) and stored in the HI register.
MSZL rS, rT	Signed 16-bit Multiply-Negate to LO LO ← 0 - rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is negated (subtracted from zero) and stored in the LO register.

MSBH rS, rT	Signed 16-bit Multiply-Subtract from HI HI ← HI - rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is subtracted from HI, ignoring any overflow. The result is stored in the HI register.
MSBL rS, rT	Signed 16-bit Multiply-Subtract from LO LO ← LO - rS * rT The contents of rS[15:0] are multiplied by rT[15:0], treating the operands as signed 2's complement values. The 32-bit product is subtracted from LO, ignoring any overflow. The result is stored in the LO register.

Table 16: 32-Bit Multiply-Accumulate Instructions

MAD rS, rT	Signed 32-bit Multiply-Accumulate t ← { HI, LO} + rS * rT LO ← t[31:0] HI ← t[63:32] The contents of rS are multiplied by rT, treating the operands as signed 2's complement values. The 64-bit product is added to the concatenation HI and LO to form a 64-bit result ignoring any overflow. The upper 32-bits of the 64-bit result are stored in the HI register. The lower 32-bits are stored in the LO register.
MADU rS, rT	Unsigned 32-bit Multiply-Accumulate t ← { HI, LO } + rS * rT LO ← t[31:0] HI ← t[63:32] The contents of rS are multiplied by rT, treating the operands as unsigned values. The 64-bit product is added to the concatenation HI and LO to form a 64-bit result ignoring any overflow. The upper 32-bits of the 64-bit result are stored in the HI register. The lower 32-bits are stored in the LO register.
MSUB rS, rT	Signed 32-bit Multiply-Subtract t ← { HI, LO } - rS * rT LO ← t[31:0] HI ← t[63:32] The contents of rS are multiplied by rT, treating the operands as signed 2's complement values. The 64-bit product is subtracted from the concatenation HI and LO to form a 64-bit result ignoring any overflow. The upper 32-bits of the 64-bit result are stored in the HI register. The lower 32-bits are stored in the LO register.

MSUBU rS, rT	Unsigned 32-bit Multiply-Subtract t ← { HI, LO } - rS * rT LO ← t[31:0] HI ← t[63:32] The contents of rS are multiplied by rT, treating the operands as unsigned values. The 64-bit product is subtracted from the concatenation HI and LO to form a 64-bit result ignoring any overflow. The upper 32-bits of the 64-bit result are stored in the HI register. The lower 32-bits are stored in the LO register.

Notes:

See Section A.3, LEXOP Instructions, for instruction encoding.

The upper 16 bits of both operand registers are ignored by 16-bit multiply instructions.

The MxxH and MxxL instructions can be freely interleaved. That is, adds and subtracts from either the HI or LO registers can be combined in a sequence with the two registers functioning “in parallel.”

The MxZx instructions can be used as stand-alone 16-bit signed multiply instructions.

4.3. MAC Pipeline

The processor may stall if a new MAC instruction is executed while a prior MAC operation is pending. Table 58 on page 113 indicates the number of cycles that must be present between MAC instructions to avoid stalls.

The MAC continues execution of multiply/accumulate arithmetic during cycles where the HOLD signal is asserted. The MAC tracks the state of the processor pipeline, and does not commit data into architectural state (the accumulators) until the corresponding processor instruction makes an M to W stage transition. Therefore, in the presence of complementary stalls (that is, stalls which occur in cycles when a MAC instruction has passed a valid E cycle), the number of stalls incurred by the MAC will be *less* than the number stated in Table 58.

The MADH, MADL, MAZH, MAZL, MSBH, MSBL, MSZH and MSZL instructions, which have 16-bit operands, are pipelined with single cycle throughput and 3 cycle latency.

The MSxx instructions are implemented by negating the multiplier for the 16-bit multiplication but are otherwise identical to the corresponding MAxx instructions. This results in subtracting the product of the original operands from the accumulator.

The MULT, MAD and MSUB instructions, which have 32-bit operands, use the same hardware in an iterative fashion to generate the 64-bit result, with 4 cycle latency for both the low and high order 32 results bits.

The HI and LO registers are used as two independent 32-bit accumulators for the 16-bit multiply instructions or as a paired 64-bit result for the 32-bit multiply instructions.

The pipelining for each category of multiplier instruction is shown below. The pipeline stages specific to the MAC are: M1, M2, M3 and WA for the 32-bit instructions, and E, C, Acc and WA for the 16-bit instructions. Note that the WA stage is where the results are valid (either in the accumulator or a hold register). The ALU pipeline is shown for comparison.

```
ALU:                                                  I D S E A M W
MULT,MULTU,MAD,MADU,MSUB,MSUBU:                       I D S E M1 M2 M3 WA
MADH,MADL,MAZH,MAZL,MSBH,MSBL,MSZH,MSZL,MTHI,MTLO:    I D S E C Acc WA
MFHI,MFLO:                                            I D S E
```

Notes:

There is no indication of overflow for the 32-bit add portion of the 16-bit multiply-accumulate instructions.

The MFHI(LO) instruction will stall the pipeline until the results of the most recent instruction which stores into HI(LO) has completed.

4.4. Accessing HI and LO after multiply instructions

The MFLO and MFHI instructions read the contents of the LO or HI registers during the E cycle of the pipeline. The following descriptions indicate how the latency of the multiply instructions affects the usage of these instructions. The most efficient sequence is shown. If the MFHI or MFLO instruction is coded earlier, the correct result will still be obtained because the hardware will stall the instruction in the E-cycle until the result is valid.

In order to prevent processor deadlock, the MAC will speculatively complete all of its arithmetic computation, even if processor HOLD is active, once it has received its operands and instruction in a valid E cycle. Initially, the simplest pipeline diagrams (assuming no stalls) are shown. Then, for clarity, some relevant stall cases are shown.

MULT (or MAD) followed by MFHI or MFLO (no stalls)

```
   MULTx       I   D   S   E   M1  M2  M3  WA
   any op          I   D   S   E   A   M   W
   any op              I   D   S   E   A   M   W
   any op                  I   D   S   E   A   M   W
   MFLO                        I   D   S   E   A   M   W
or MFHI                        I   D   S   E   A   M   W
```

MADH followed by MFHI (no stalls)

```
   MAZH0       I   D   S   E   C   Acc WA
   MADH1           I   D   S   E   C   Acc WA
   MADH2               I   D   S   E   C   Acc WA
   MADH3                   I   D   S   E   C   Acc WA
   any op                      I   D   S   E   A   M   W
   any op                          I   D   S   E   A   M   W
   MFHI                                I   D   S   E   A   M  W
   HI contains                         A0  A1  A2  A3
```

In the presence of stalls, the MAC completes its normal sequence of operations prior to WA. If the pipeline state of the corresponding instruction (shown below as MAD-pipe) is not ready to complete an M to W stage transition, the data is held in either the A-stage or M-stage registers, depending on the pipeline state. The HOLD line is assumed to be a stall caused by some purpose other than the MAC (e.g., instruction cache miss).

MULT (or MAD) followed by MFHI or MFLO (MAC stall only)

```
MAD          I   D   S   E   M1  M2  M3  A_R M_R WA
MAD-pipe     I   D   S   E   A   A   A   A   M   W
MFLO             I   D   S   E   E   E   E   A   M   W
MAC-HOLD                     X   X   X
    HOLD
```

MULT (or MAD) followed by MFHI or MFLO (MAC stall concurrent with other stall)

```
MAD          I   D   S   E   M1  M2  M3  A_R A_R M_R WA
MAD-pipe     I   D   S   E   A   A   A   A   A   M   W
MFLO             I   D   S   E   E   E   E   E   A   M   W
MAC-HOLD                     X   X   X
    HOLD                     X   X   X   X
```

4.5. Divider Overview and Register Usage

Given a dividend DEND, and divisor DVSR, the divider generates a quotient QUOT (stored in LO) and remainder REM (stored in HI) that satisfy the following conditions, regardless of the signs of DEND and DVSR:

```
DEND = DVSR * QUOT + REM,
0 <= abs(REM) < abs(DVSR)
```

where REM and DEND have the same sign.

It is worth noting that the requirement that REM and DEND have the same sign is not universally accepted if DEND and DVSR are not both positive. (For example the Modula-3 language expects -5DIV3=-2 and -5MOD3=+1, whereas the divider generates QUOT=-1, REM=-2 in agreement with FORTRAN and others.) These examples show the possible combinations of signs:

DEND	DVSR	QUOT	REM
+19	+5	+3	+4
-19	+5	-3	-4
+19	-5	-3	+4
-19	-5	+3	-4

The divider is an iterative circuit that generates 2 quotient bits per cycle, with an additional 3 cycles required due to pipelining considerations.

Thus the pipeline flow of a division instruction and the most efficient subsequent read of the quotient (using MFLO) is as shown in the following diagram. If the MFLO is issued earlier it will stall until the divide completes. Less than 19 instructions may be issued if some of them take more than one cycle to complete (due to cache misses or data dependent stalls, for example).

```
DIV   I   D   S   E   D0  D1  D2  ...  D16 D17 D18
...
18 cycles
...
MFLO                                   I   S   E   A   M   W
```

5. Memory Management Unit (MMU)

This section discusses the LX4380's optional MMU. The MMU incorporates the following features:

- 4KB or 64KB memory pages, with 32-bit virtual and physical addresses and a 6-bit Address Space Identifier (ASID).
- The 4GB address space is broken up into 2GB mapped user space (kuseg), 0.5GB unmapped, cacheable kernel space (kseg0), 0.5GB unmapped, uncacheable kernel space (kseg1), and just less than 1GB of mapped kernel space (kseg2). The remaining 16 MB address space above kseg2 (16 MB) is dedicated to specific debug devices (upper-kseg2).
- Configurable joint Translation Lookaside Buffer (TLB): 16, 32 or 64 entries.
- Hardware-managed Instruction TLB (ITLB) and Data TLB (DTLB) caches, each with three entries.
- ENTRYHI, ENTRYLO, CONTEXT and INDEX registers.
- Read-only RANDOM register, Read-only WIRED register.
- The MMU signals the appropriate User TLB (UTLB) exception (TLBL or TLBS) or the appropriate BEV0/BEV1 exception (TLBL, TLBS, TLBMOD).
- Implements the TLBP, TLBR, TLBWI, TLBWR instructions.

5.1. Memory Regions

Table 17: MMU Address Translation

Region Name	Virtual Address	Physical Address	Cacheability	Permission	Writable
kuseg	0x0000_0000-0x7fff_ffff	mapped via TLB	set via TLB	user	set via TLB
kseg0	0x8000_0000-0x9fff_ffff	0x0000_0000-0x1fff_ffff	cacheable	kernel	yes
kseg1	0xa000_0000-0xbfff_ffff	0x0000_0000-0x1fff_ffff	uncacheable	kernel	yes
kseg2	0xc000_0000-0xfeff_ffff	mapped via TLB	set via TLB	kernel	set via TLB
upper-kseg2	0xff00_0000-0xffff_ffff	0xff00_0000-0xffff_ffff	uncacheable	kernel	yes

5.2. Registers

The registers described in this section are accessed with the MFC0 and MTC0 operations. The 0 fields in these registers are ignored on write and are 0 on read. For compatibility with future LX4380 versions, they should be written with 0.

TLB Entry

A TLB Entry is a 64-bit quantity that stores a Virtual to Physical Memory Mapping. The fields of a TLB Entry are identical to the fields of the ENTRYHI and ENTRYLO registers. The TLB entries are written with the TLB Write Indexed (TLBWI) and TLB Write Random (TLBWR) instructions. They are read with the TLB Read (TLBR) instruction. The entries can be examined for a match to a specific Virtual Page Number (VPN), Address Space ID (ASID) pair using the TLB Probe (TLBP) instruction. The MMU also compares TLB entries to virtual instruction or data memory accesses to determine if a match exists.

ENTRYHI: Coprocessor 0 General Register Address = 10

31 - 12	11 - 6	5 - 0
VPN	ASID	000000

Field	Description	R/W	Reset
VPN	Virtual Page Number. Low order 4 bits must be set to zero if MMU is configured for 64KB pages.	R/W	unmapped[a]
ASID	Address Space ID.	R/W	0

a. VPN is reset to an unmapped address that depends on the page size configuration: 0x80000 for 4KB pages, 0x8000 for 64KB pages.

ENTRYHI is composed of two fields; the VPN and ASID. When TLB Probe (TLBP) instruction is executed, the MMU compares the VPN and ASID fields of the ENTRYHI register (but also accounting for the Global bit of the TLB entry) to each entry in the TLB. The comparison mechanism is described in Section 5.3.

When the processor executes a TLB Read (TLBR) instruction, the TLB entry specified by the INDEX register is loaded into ENTRYHI and ENTRYLO. If the TLB is configured with a >4KB page size, the corresponding lower order bits of the VPN in the ENTRYHI register will always return zeroes.

The VPN and ASID fields of ENTRYHI are written into the TLB when the CPU executes the TLB Write Index (TLBWI) or TLB Write Random (TLBWR) instruction.

ENTRYLO: Coprocessor 0 General Register Address = 2

31 -12	11	10	9	8	7 - 0
PFN	N	D	V	G	00000000

Field	Description	R/W	Reset
PFN	Page Frame Number. Low order 4 bits must be set to zero if MMU is configured for 64KB pages.	R/W	0
N	Noncacheable.	R/W	0
D	Dirty.	R/W	0
V	Valid	R/W	0
G	Global	R/W	0

ENTRYLO is composed of the following fields: Page Frame Number (PFN), Noncacheable (N), Dirty (D), Valid (V) and Global (G).

The PFN, N, D, V and G fields of ENTRYLO are written to a TLB entry when the CPU executes the TLBWI or TLBWR instructions. During any mapped instruction or data access and during the execution of a TLBP instruction, the MMU examines the G bit of the TLB entry to determine if a VPN, ASID combination was a match. A mapped instruction or data access with a TLB match (based on VPN, ASID and G) will examine the D and V bits to signal an exception, as described in Section 5.6 on page 42. The N bit is used to determine whether the access is cacheable.

When the processor executes a TLB Read (TLBR) instruction, the TLB entry specified by the INDEX register is loaded into ENTRYHI and ENTRYLO. If the TLB is configured with a 64KB page size, the corresponding lower order bits of the PFN in the ENTRYLO register returns zeroes.

INDEX: Coprocessor 0 General Register Address = 0

31	30 - 14	13 - 8	7 - 0
P	00000000000000000	INDEX	00000000

Field	Description	R/W	Reset
P	Probe failure - 1 indicates TLBP instruction did not find TLB match.	R	0
INDEX	Result of TLBP instruction; TLB entry to write with TLBWI instruction,	R/W	0

The INDEX register is used by the TLBP, TLBR and TLBWI instructions, as described in Section 5.3.

The INDEX register contains a 6-bit address into the TLB (INDEX), and a one-bit Probe failure field (P).

RANDOM: Coprocessor 0 General Register Address = 1

31 -14	13 - 8	7 - 0
00000000000000000	RANDOM	00000000

Field	Description	R/W	Reset
RANDOM	Decremented with every valid M-stage instruction completion.	R	TLB_MAX[a]

a. TLB_MAX is defined by the TLB_ENTRIES *lconfig* setting.

The RANDOM register contains a 6-bit address into the TLB that is decremented on every valid M-stage instruction completion (consistent with R4000 operation). The address decrements from the largest TLB index available on the processor (15, 31 or 63), down to the value of the WIRED register. The address then wraps back to the largest TLB index. This ensures that the lowest WIRED entries (0 to WIRED-1) entries are not randomly overwritten.

When the processor executes a TLB Write Random (TLBWR) instruction, the processor writes the TLB entry that is identified by RANDOM. However, the processor selects a different entry for writing if RANDOM identifies an entry that was written by either of the last two executions of TLBWR. This ensures that thrashing does not occur in the event of back-to-back TLB misses.

WIRED: Coprocessor 0 General Register Address = 6

31 - 6	5 - 0
00000000000000000000000000	WIRED

Field	Description	R/W	Reset
WIRED	Number of wired TLB entries. This field is a read-only constant.	R	4 or 8[a]

a. Reset value depends on the *lconfig* TLB_ENTRIES setting: 4 if 16 or 32 entries, 8 if 64 entries.

The WIRED register contains a 6-bit address of the lowest entry that may be overwritten by the TLB Write Random (TLBWR) instruction.

CONTEXT: Coprocessor 0 General Register Address = 4

31 - 21	20 - 2	1 - 0
PTEBase	BadVPN	00

Field	Description	R/W	Reset
PTEBase	Page Table Entry Base address	R/W	0
BadVPN	Bad Virtual Page Number.	R	0

The CONTEXT register contains a pointer to the base of a virtual page table, and a copy of bits 30:12 of the BADVADDR register. It may be used by kernel software to accelerate page table lookup operations.

5.3. TLB Instructions

These opcodes for these instructions are identical to those used in the MIPS-I (R2000/R3000) instruction set.

Instruction	Description
TLBP	TLB Probe INDEX ← index of TLB entry which matches ENTRYHI[VPN,ASID] Examines each TLB entry to determine if it matches the values contained in ENTRYHI. The determination of a match for entry i is: match[i] = (ENTRYHI[VPN] == TLBHi[i][VPN]) && TLBLo[i][GLOBAL] || (ENTRYHI[ASID] == TLBHi[i][ASID])) where TLBHi and TLBLo are the appropriate fields of the TLB entry. If no TLB match is found, the INDEX register P bit is set to 1, and the INDEX field is undefined. If a single TLB match is found, the INDEX register P bit is set to 0, and the INDEX field contains the 6-bit address of the matching entry. If multiple TLB matches are found, the INDEX register P bit and INDEX value are both undefined. (This could only arise because of an error in OS kernel code.)
TLBR	TLB Read { ENTRYHI, ENTRYLO } ← TLB[INDEX] Updates the ENTRYHI and ENTRYLO registers with the contents of the TLB entry specified by the INDEX register. If the value of INDEX register is greater than the number of TLB entries implemented, the TLB Read instruction returns a 32-bit zero result to both the ENTRYHI and ENTRYLO registers.
TLBWI	TLB Write Indexed TLB[INDEX] ← { ENTRYHI, ENTRYLO } Updates the entry of the TLB specified by the INDEX field of the INDEX register, with the values specified in ENTRYHI and ENTRYLO. If the value of INDEX is greater than the number of registers implemented, no entry is updated.
TLBWR	TLB Write Random TLB[RANDOM] ← { ENTRYHI, ENTRYLO } Updates the entry of the TLB specified by the RANDOM field of the RANDOM register, with the values specified in ENTRYHI and ENTRYLO.

5.4. Mapped Address Translation

Mapped address translation is performed for instruction fetches, data loads, data stores and the CACHE instruction.

Any memory reference to kseg0, kseg1, or upper-kseg2 is an unmapped access. An unmapped access does not generate a TLB translation; instead, the direct translation to the PFN is used, as shown in Table 17. (Note:

this specifies both the physical location and cacheability of the request. All of the unmapped areas are assumed writable.)

If the reference is to a kuseg or kseg2 address, a TLB translation is performed by checking all TLB entries for a match, similar to the TLBP instruction described in Section 5.3.

The list below shows the sequence of decisions and actions performed by the MMU. Note that the VPN comes from the address being tested, while the ASID is held in ENTRYHI.

- If the processor is in user mode, and the reference is to kseg2, generate an AdEL or AdES exception. No translation is performed.
- If there is no TLB match and the reference is to kuseg, a UTLB TLBL or TLBS (i.e. miss) exception is taken.
- If there is no TLB match and the reference is to kseg2, a BEV TLBL or TLBS (i.e. miss) exception is taken.
- If there is a TLB match and the matching entry is not valid, a BEV TLBL or TLBS (i.e. miss) exception is taken.
- If the request is for a store data operation and the Dirty bit is not set, a BEV Mod exception is taken.
- If the N field is zero, a cacheable request is made.
- If the N field is one, a noncacheable request is made.

5.5. Stalls

See Section D.8 on page 114 for information on stalls that are caused by the MMU.

5.6. MMU Exceptions

The User TLB (UTLB) exceptions go to a unique exception vector that can utilize a very low level of functionality for user applications only, while the more complete BEV (Boot Exception Vector) versions include handling for privileged (OS) tasks, invalid entries and access violations (modify exceptions).

Table 18: TLB Exceptions

Exception Type	BEV0 Exception Address	BEV1 Exception Address	CAUSE ExcCode
UTLB TLBL	0x8000_0000	0xbfc0_0100	2
UTLB TLBS	0x8000_0000	0xbfc0_0100	3
BEV TLBL	0x8000_0080	0xbfc0_0180	2
BEV TLBS	0x8000_0080	0xbfc0_0180	3
BEV TLBMOD	0x8000_0080	0xbfc0_0180	1

5.7. Instruction and Data TLBs

The Instruction TLB (ITLB) and Data TLB (DTLB) are hardware-managed caches of the most recent TLB translations for instruction fetches (ITLB) and loads or stores (DTLB). Each TLB cache contains three entries, which improves processor performance when using the MMU. See the Stall Table Appendix for a description of the stall cycles associated with the ITLB and DTLB.

5.8. Comparison of LX4380 MMU features

This section summaries the differences between LX4380 MMU and the R3000 MMU.

- The LX4380 supports the RANDOM register. However, it decrements according to the R4000 implementation - once for every valid M-stage instruction completion.
- The RANDOM register does not select the last two randomly written TLB entries to avoid deadlock/livelock conditions that could otherwise occur.
- The LX4380 supports the WIRED register (an R4000 register). However, it is read-only and its value is based on the number of TLB entries.
- The LX4380 provides a configurable number of TLB entries (16, 32 or 64). The R3000 provides 64 entries.
- The regions of memory that are mapped/unmapped differ from the R3000. The LX4380 does not map the upper 16MB above kseg2 (0xFF*). This region is unmapped and uncached, and is identified as upper-kseg2.

5.9. Use Restrictions After TLB Modification

When a TLBWI or TLBWR instruction is used to write an entry in the TLB, the kernel must ensure that the new entry is not used until the write takes effect. This can be done by performing the TLB write sufficiently in advance of the return from TLB exception.

The following list documents the number of *instruction cycles* that must elapse before an updated TLB entry is used for the indicated purpose. (An instruction cycle is any coded instruction, as well as *issue stalls* that are generated by the processor. *Pipeline stalls* do not count as instruction cycles. See Section D.1.)

Ifetch (entry added to TLB)	5 cycles
Ifetch (entry removed trom TLB)	6 cycles
Load/Store/Cache	3 cycles
TLBP	3 cycles
TLBR	2 cycles

Consider an example involving a UTLBL exception. The exception handler must add the required entry to the TLB. According to the above list, 5 instruction cycles are required after the TLB write for an instruction fetch to properly fetch from the newly added page. The following code sequence is shown as an example of a valid write of the TLB entry needed by the target of "jr k0". The instruction cycles are counted for clarification:

```
tlbwr                       (TLBW* has a 1 cycle issue stall)
<tlbwr issue stall>         <- +1
jr          k0              <- +1   (JR has a 2 cycle issue stall after delay slot)
rfe                         <- +1
<jr issue stall #1>         <- +1
<jr issue stall #2>         <- +1
<New Ifetched Page here>
```

5.10. Use Restrictions After ASID Modification

The current ASID that is used in the processor is held in the ENTRYHI Coprocessor 0 register. Any write to this register can change the current ASID. The ASID should only be modified when executing in TLB unmapped space.

When changing the ASID, the operating system's kernel must ensure that any operation that depends on the ASID is not executed until the change has taken affect. The following list describes the number of *instruction cycles* that must elapse before the updated ASID can be used for the intended purpose. Only a MTC0 to the ENTRYHI register or a TLBR instruction can change the current ASID.

Ifetch	5 cycles
MFC0	2 cycles
Load/Store/Cache	3 cycles
TLBP	2 cycles
TLBW	1 cycle

On a TLBL exception, it is possible that the ASID may be set just before returning to a process running in TLB mapped space. According to the above list, 5 instruction cycles are required after the ASID update for an Ifetch to acknowledge the update to the ASID. The following code sequence is shown as an example of a valid update to the current ASID and return from the exception handler. The instruction cycles are counted for clarification:

```
mtc0    k1, C0_ENTRYHI      (MTC0 has a 2 cycle issue stall)
<mtc0 issue stall #1>       <- +1
<mtc0 issue stall #2>       <- +1
jr          k0              <- +1 (JR has a 2 cycle issue stall after delay slot)
rfe                         <- +1
<jr issue stall #1>         <- +1
<jr issue stall #2>         <- +1
<New Ifetched Page Here>
```

5.11. Determining the Number of TLB Entries via Software

To determine the number of TLB entries that are present, software can exploit the fact that the TLB Read (TLBR) instruction returns zero if a non-existent TLB entry is read. Software may write a non-zero entry to a TLB location using the TLB Write Indexed (TLBWI) instruction, and examine the result of a subsequent TLBR instruction to determine if the TLB entry actually exists. The non-zero entry should be constructed from an unmapped VPN (e.g. in kseg0) to ensure consistent operation of the processor.

6. Coprocessor Interface

The LX4380 processor provides Coprocessor Interfaces (CIs) for the attachment of application-specific coprocessors. This section provides a description of these access points.

6.1. Attaching a Coprocessor Using the Coprocessor Interface (CI)

A coprocessor may contain up to 32 general registers and up to 32 control registers. Each of these registers is up to 32 bits wide. Typically, programs use the general registers for loading and storing data on which the coprocessor operates. Data is moved to the coprocessor's general registers from the processor's general registers with the MTCz instruction. Data is moved from the coprocessor's general registers to the processor's general registers with the MFCz instruction. Main memory data is loaded into or stored from the coprocessor's general registers with the LWCz and SWCz instructions.

Programs may load and store the coprocessor's control registers from the processor's general registers with the CTCz and CFCz instructions respectively. Programs may not load or store the control registers directly from main memory.

The coprocessor may also provide a condition flag to the processor. The condition flag is tested with the BCzT and BCzF instructions. These instructions indicate that the program should branch if the condition is true (BCzT) or false (BCzF).

6.2. Coprocessor Interface (CI) Signals

The CI provides the mechanism to attach a custom coprocessor to the processor. The CI snoops the instruction bus for coprocessor instructions and then gives the coprocessor the signals necessary for reading or writing the general and control registers. I/O is relative to the LX4380 CI.

Table 19: Coprocessor Interface Signals

Signal	I/O	Description
Czcondin	input	Branch flag.
Czrd_addr[4:0]	output	Read address.
Czrhold	output	Coprocessor must stall when asserted (one).
Czrd_gen	output	General register read command.
Czrd_con	output	Control register read command.
Czrd_data[31:0]	input	Read data.
Czwr_addr[4:0]	output	Write address.
Czwr_gen	output	General register write command.
Czwr_con	output	Control register write command.

Signal	I/O	Description
Czwr_data[31:0]	output	Write data.
Czinvld_M	output	One indicates invalid instruction in M stage.
Czxcpn_M	output	Exception flag, one indicates exception in M stage.

In the above table, z indicates a user coprocessor number (1 or 2). The addresses, output data, and control signals are supplied to the application's coprocessor on the rising edge of the system clock.

6.3. Coprocessor Write Operations

During a coprocessor write, the CI sends Czwr_addr and Czwr_data, and asserts either Czwr_gen or Czwr_con. The coprocessor write operations are subject to a pipeline hold. That is, if either of the write control signals is asserted while Czrhold is asserted, the coprocessor must defer the write to the appropriate register on the subsequent rising edge of the clock. The target register is a decoding of Czwr_addr, Czwr_gen and Czwr_con. The LWCz, MTCz, and CTCz instructions cause a coprocessor write.

Figure 4 illustrates two coprocessor write operations. The operation labeled A does not encounter a pipeline hold. The operation labeled B encounters a pipeline hold that lasts one cycle.

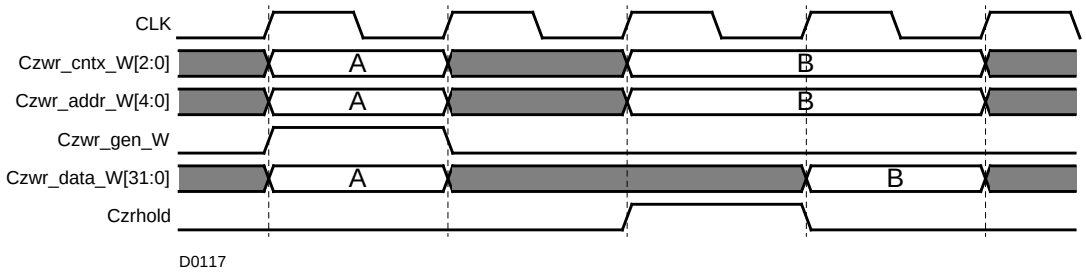


Figure 4: Coprocessor Write

6.4. Coprocessor Read Operations

During a coprocessor read, the CI sends Czrd_addr and asserts either Czrd_gen or Czrd_con. The coprocessor must return valid data through Czrd_data in the following clock cycle. If the processor asserts Czrhold, indicating that it is not ready to accept the coprocessor data, the coprocessor must hold the previous value of Czrd_data. The target register for the read is a decoding of Czrd_addr, Czrd_gen, and Czrd_con. The instructions causing a coprocessor read are SWCz, MFCz, and CFCz.

Figure 5 illustrates three coprocessor read operations. The signal names beginning with Cz_stage_ represent application specific signals in a coprocessor design and are shown to illustrate the pipelining within a coprocessor. Coprocessor designs that perform internal operations as a result of a read must include such stages in their read logic to allow for the cancellation of a coprocessor read that could arise from an exception that is encountered during an earlier instruction. (This is possible, for example, if the coprocessor implements a read FIFO See Figure 6 and Figure 7 for examples of instruction cancellations).

In the example of Figure 5, coprocessor read operation A encounters a pipeline hold while in the A stage. Operations A, B and C encounter a pipeline hold while in the E, A and W stages respectively. Although not

shown in the diagram, coprocessor operations can also be held while they are in the M stage.

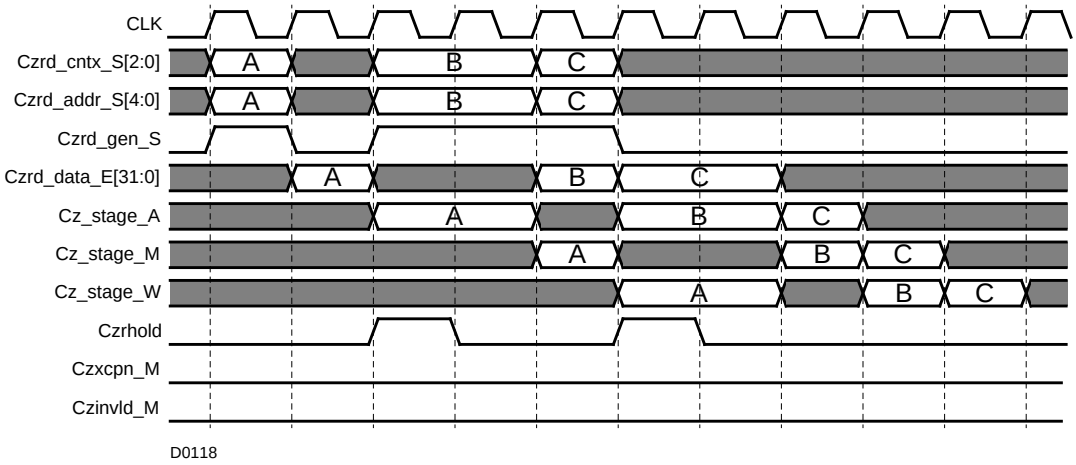


Figure 5: Coprocessor Read

The CPU stalls the pipeline so that the program can access data read by a coprocessor instruction in the immediately following instruction. For example, if an MFCz instruction reads data from the coprocessor and stores it in the processor's general register $4, the program can get access to that data in the following instruction:

```
mfc2    $4, $3      # Move from COP2 to CPU register $4
subu    $5, $4, $2  # Subtract $R2 from $R4 and store in $5
```

When the processor initiates a coprocessor read, the coprocessor must return valid data in the following clock cycle. The coprocessor cannot stall the CPU. Applications must ensure that the source code does not access invalid coprocessor data if the coprocessor operations take several clock cycles to complete. This is done in one of three ways:

- Ensure that software does not access data from the coprocessor until N instructions after the coprocessor operation has started. This is the least desirable method as it depends on the relative execution of the processor and coprocessor. It can also complicate software debug.

- Have the coprocessor send an interrupt to the processor, and the service routine for that interrupt accesses the appropriate coprocessor registers.

- Have the coprocessor set the Czcondin flag when its operation is complete. The source code can poll the flag as shown in the example below:

```
        mtc2    $2, $3      # store data to COP2 general register $3
        ctc2    $3, $5      # set COP2 control register $5 to start
        nop
loop:   bc2f    loop        # branch back to loop if Czcondin bit off
        nop                 # branch delay slot
        mfc2    $4, $7      # get results from COP2 general register $7
```

6.5. Coprocessor Interface and Pipeline Stages

Coprocessor writes occur in the W stage of the instruction pipeline. For coprocessor reads, the processor generates address, rd_gen, and rd_con signals during the E stage, and the coprocessor returns data during the

A stage which is passed by the CI to the processor in the M stage. The processor introduces two pipeline bubbles after coprocessor instructions to ensure that the result of a MTCz instruction can be used by the immediately following instruction.

```
mtc2            I D S E A M W
bubble 1          I D S E A M W
bubble 2            I D S E A M W
mfc2                  I D S E A M W

wr_gen (W)                  X
rd_gen (E)                  X
rd_data (A)                   X
```

6.5.1. Pipeline Holds

The Czwr_addr, Czwr_data, Czwr_gen and Czwr_con signals need not be registered. The coprocessor may decode these W stage signals directly to the appropriate register. However, the coprocessor must ignore the assertion of the write control signals when Czrhold is asserted. See Figure 4.

The coprocessor must register the read address and the control signals Czrd_gen and Czrd_con. It must hold the A stage registered values of these signals when Czrhold is active high, and should make the read data output a function of the A stage registered read address and control signals. If the coprocessor includes additional internal stages that perform actions as a result of a read operation, they must also be held by the Czrhold signal. See Figure 5.

6.5.2. Pipeline Invalidation

Under certain circumstances the instruction pipeline can contain an instruction that must be discarded. This may be due to mispredicted branches, cache misses, exceptions, inserted pipeline bubbles etc. In such cases, the CI may decode an instruction that must actually be discarded.

For the coprocessor write-type instructions, the CI will only issue the W stage control signals Czwr_gen and Czwr_con for valid instructions. The coprocessor does not need to qualify these controls.

For the coprocessor read-type instructions, the CI may issue the E stage control signals Czrd_gen and Czrd_con for instructions that must be discarded. If the coprocessor can tolerate speculative reads then it need not qualify those signals. However, if the coprocessor performs “destructive” reads, such as updating a FIFO pointer upon read, then it must use the qualifying signals Czxcpn_m and Czinvld_m as follows:

When the Czxcpn_M signal is asserted by the processor, the coprocessor must discard any S, E and A stage operations, even if Czrhold is also asserted. This Czxcpn_M signal indicates that preceding instruction in the M stage of the processor pipeline has taken an exception and that subsequent instructions in the pipeline must be discarded. Figure 6 illustrates the occurrence of an exception while a coprocessor instruction is at the S, E or A stages.

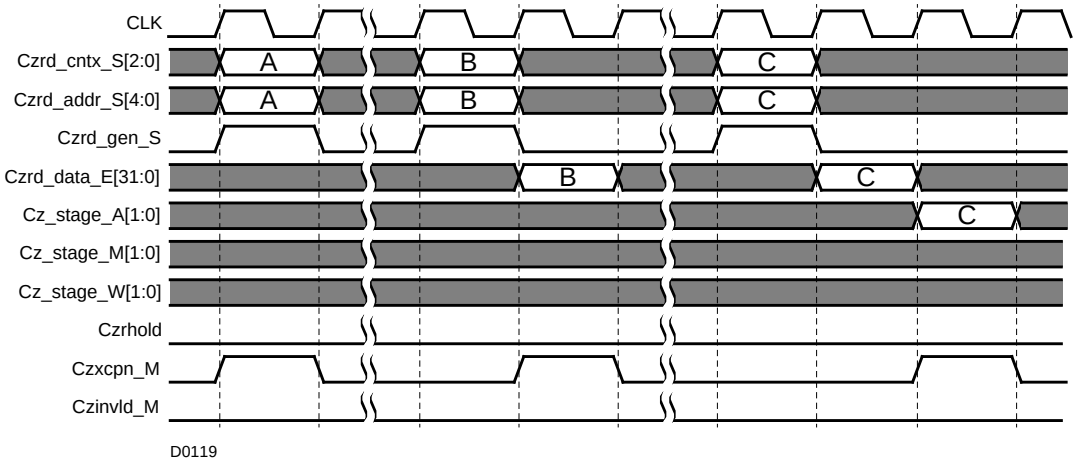


Figure 6: Exception During Coprocessor Read

The processor asserts Czinvld_M signal to invalidate the instruction in the M stage. If the coprocessor cannot tolerate speculative reads, it must tentatively compute its E, A and M stage results for any read operation. If Czinvld_M is asserted when the read operation is in the M stage (including any period when Czrohold is asserted), then the coprocessor must discard the tentative results. If the read operation passes advances to the W stage without the assertion of Czinvld_M, then the coprocessor must commit its temporary results. An example of an invalidated read operation is shown in Figure 7.

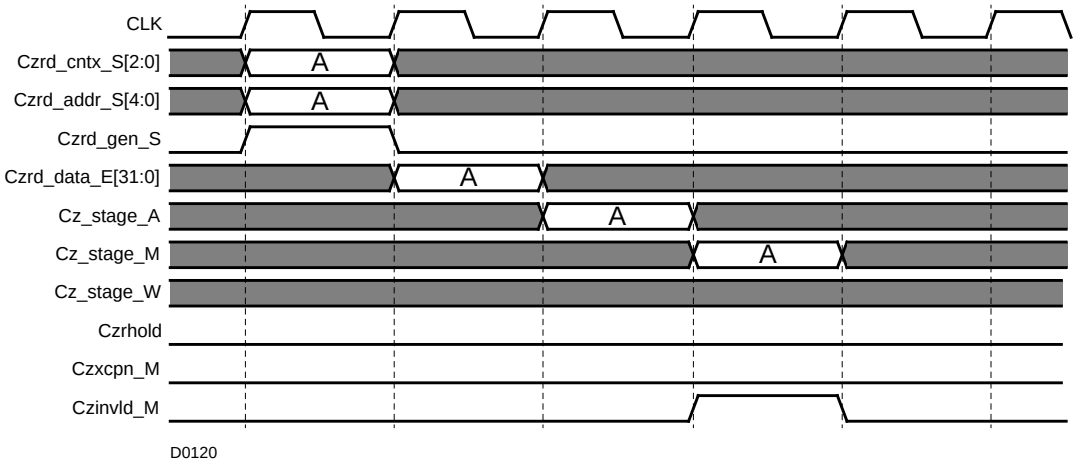


Figure 7: Invalidation of Coprocessor Read

7. Local Memory

7.1. Local Memory Overview

This section describes how memories are configured and connected to the LX4380 using the Local Memory Interfaces (LMIs). This section provides a brief summary of the conventions and supported memories. Section 7.2 describes the control register that allows software control over certain aspects of the LMIs. The subsequent sections cover each of the LMIs in detail.

This section also discusses configuration options and the ports that customers must access to connect application specific RAMs that are used by the LX4380 LMIs. All of the signals between the LMIs and RAMs are automatically configured by *lconfig*, the LX4380 configuration tool. *Lconfig* also produces documentation of the exact RAMs required for the chosen configuration settings, and generates RAM models used for RTL simulation.

The LMIs connect to RAMs that service the LX4380 processor's local instruction and data busses. The LMIs also provide the pathways from the processor to the system bus. The LX4380 includes an LMI for each of the local memory types. The sizes of the RAMs are customer selectable. The LX4380 LMIs directly support synchronous RAMs that register the address, write data, and control signals at the RAM inputs. The LMIs also supply redundant read enable and chip select lines for each RAM, which may be required for some RAM types.

Lexra supplies an integration layer for the LMIs and the memory devices connected to them. In this layer, memory devices are instanced as generic modules satisfying the depth and width requirements for each specific memory instance. The *lconfig* utility supplies a summary of the memory devices required for the chosen configuration. In most cases, customers simply need to write a wrapper that connects the generic module port list to a technology specific RAM instance inside the RAM wrapper.

The LX4380 is configurable for a 16, 32, 64, or 128-byte cache line size. The tag store RAM sizes shown in the tables of this section assume a *32-byte line size*. The documentation produced by *lconfig* indicates the required tag RAMs for the selected configuration options, including the line size. As a general rule, a doubling of the line size results in halving the tag store depth.

The valid bits within tag stores are automatically cleared by the LMIs upon reset. The data cache implements write-through or write-back protocols, selectable with *lconfig*. Caches do not snoop the system bus. The LX4380 uses RAMs with byte write granularity for its data stores. Byte write granularity results in more efficient operation of store byte and store half-word instructions.

The LMIs use physical addresses for all operations. Caches are physically indexed and store physical tags.

Table 20 summarizes the local memories that can be integrated with the LX4380.

Table 20: Local Memory Interface Modules

Name	Description
ICACHE	Direct mapped or two-way set associative instruction cache.
IMEM	Instruction RAM.
DCACHE	Direct mapped or two-way set associative data cache.
DMEM	Data RAM.

7.2. Cache Control Register: CCTL

CCTL. CP0 General Register Address = 20

31-12	11	10	9	8	7-6	5	4	3-2	1	0
Rsrvd	DMEMOff	DMEMOn	DWBInval	DWB	Rsrvd	IMEMOff	IMEMFill	ILock	IInval	DInval

When reading this register, the contents of the Reserved bits are undefined. When writing this register, the contents of the Reserved bits should be preserved.

The IMEMFill and IMEMOff bits of the CCTL register control the contents and use of any local IMEM memory configured into the LX4380. When the LX4380 is reset, the LMI clears an internal register to indicate that the entire IMEM LMI contents are invalid. When IMEM is invalid, all cacheable fetches from the IMEM region will be serviced by the instruction cache, if an instruction cache is present.

A transition from 0 to 1 on IMEMFill causes the LMI to initiate a series of line read operations to fill the IMEM contents. The addresses used for these reads are defined by the configured BASE and TOP addresses of the IMEM, described in Section 7.5. The processor stalls while the entire IMEM contents are filled by the LMI. Thereafter, the LMI sets its internal IMEM valid bit and will service any access to the IMEM range from the local IMEM memory. The time that an IMEM fill takes to complete is the number of line reads needed to fill the IMEM range, multiplied by the latency of one line read, assuming there is no other system bus traffic.

A transition from 0 to 1 on IMEMOff causes the LMI to clear its internal IMEM valid bit. Subsequent cacheable fetches from the IMEM region will be serviced by the instruction cache. To use the IMEM again, an application must re-initialize the IMEM contents through the IMEMFill bit of the CCTL register.

A transition from 0 to 1 on DMemOff causes the Dcache LMI to disable the DMEM. Subsequent access in the DMEM region will be serviced by the data cache (cacheable addresses) or system memory (uncacheable addresses). To use the DMEM after it has been disabled, an software must cause a transition from 0 to 1 on DMemOn. This will re-enable the DMEM. The state of the DMEM will be as it was when it was disabled.

The ILock field controls set locking in the two-way set associative instruction cache. When ILock is 00, the instruction cache operates normally. When ILock is 10, “LockGather” mode, all cached instruction references are forced to occupy way 1. The hardware will invalidate lines in way 0 if necessary to accomplish this. When ILock is 11, “LockedDown” mode, lines in way 1 are never displaced – i.e. they are locked in the cache. Way 0 is used to hold other lines as needed. ILock = 01 is reserved. If this setting is used, results are undefined.

To utilize the cache locking feature, software should execute at least one pass of critical subroutines or loops with ILock set to 10. After this has been done, ILock should be set to 11 to lock the critical code into way 1,

and use way 0 for other code.

The IInval bit controls hardware invalidation of the instruction cache. A transition from 0 to 1 on IInval initiates a hardware invalidation sequence of the entire instruction cache.

The DInval, DWB and DWBInval bits control hardware invalidation of the data cache. A transition from 0 to 1 on DInval initiates a hardware invalidation sequence of the entire data cache. Any dirty lines are discarded, i.e. not written back to main memory. A transition from 0 to 1 on DWB initiates a hardware sequence to write-back all dirty lines in the data cache, leaving them in the clean state. Lines that are already clean or invalid have no operation performed. A transition from 0 to 1 on DWBInval initiates a hardware sequence to write-back all dirty lines in the data cache, and to invalidate all lines in the data cache regardless of their initial state. A simultaneous (with one MTC0 instruction) transition from 0 to 1 on more than one of DInval, DWB or DWBInval leads to unpredictable results. The DMEM, if present, is unaffected data cache CCTL operations.

The hardware invalidation sequence for the instruction and data caches requires up to four cycles per cache line to complete. When dirty data must be written back to main memory, the amount of time required is dependent on the state of the data cache and the performance of the system bus.

The LX4380 observes changes in the contents of the CCTL register in the W stage. Instructions that are in progress in earlier stages will not be affected by an instruction cache or data cache invalidation, or IMEM fill. This means, for example, that after a write to CCTL that invalidates the instruction cache, several instructions that were fetched before the invalidation may be executed, even if those instructions were invalidated from the instruction cache.

If a small number of lines known must be invalidated, it is more efficient for software to execute the CACHE instruction to affect the state of specific cache lines. This is described in the next section.

7.3. CACHE Instruction

The CACHE instruction allows software to affect the state of specific cache lines.

CACHE op, offset(rS)	Cache Operation Performs a data cache operation at address (rS + offset). An address is computed as *base + offset*, where *base* is reg rS and the *offset* is the 16-bit offset sign-extended to 32 bits. The address is translated using the SMMU or the optional MMU as for a LB instruction to form a physical address. The *op* is a 5-bit data cache operation. If the line containing the byte with the specified physical address is not found in the data cache, then no cache operation is performed regardless of the value of *op*. Otherwise the following operation is performed: 10001: Inval — the line is invalidated 10101: WBInval — the line is written back if dirty, and invalidated regardless of state 11001: WB — the line is written back if dirty, and left in the clean state. others: reserved The operation is performed even if the address falls within the address range defined for DMEM.[a] If the mapped or unmapped address translation indicates that the address of the line found in the cache is uncacheable (for example by using a kseg1 address to access a kseg0 line) it is undefined whether or not the operation specified by the instruction is performed. The execution of the CACHE instruction is subject the same address exceptions as the LB instruction, and to a Coprocessor Unusable exception under the same conditions as a coprocessor instruction that accesses CP0.

a. Memory addresses within the DMEM range might be held in the data cache if DMEM has been disabled with the DMEMOff bit in the CCTL register. This is possible even when DMEM access is re-enabled with the DMEMOn bit.

7.4. Instruction Cache (ICACHE) LMI

The ICACHE LMI supplies the interface for a direct mapped or two-way set associative instruction cache attached to the LX4380 local bus. The degree of associativity is specified through lconfig. The ICACHE LMI participates in cacheable instruction fetches, but only if the address is not claimed by the IMEM module. The configurations supported by ICACHE, and the synchronous RAMs required for each, are summarized in Table 21.

The instruction store for the two-way ICACHE consists of two 64-bit wide banks, with separate write-enable controls. The tag store consists of one RAM bank with tag and valid bits for way 0, and a second RAM for way 1 that holds the tag, valid, LRU (Least Recently Used), and lock bits. When a miss occurs in the two-way ICACHE, the LRU bit is examined to determine which way of the set to replace, with way 0 being replaced if LRU is 0, and way 1 being replaced if LRU is 1. The state of the LRU bit is then inverted. To optimize the timing of cache reads, the two-way ICACHE uses the state of the LRU bit to determine which way should be

returned to the CPU. In the following cycle, the ICACHE determines if the correct way was returned. If not, the ICACHE takes an extra cycle to return the correct element to the CPU and inverts the LRU bit.

Table 21: ICACHE Configurations

Configuration	ICACHE_INST RAM	ICACHE_TAG RAM
no instruction cache	no RAM required	no RAM required
1K bytes, 2-way	2 x 64 x 64 bits	16 x 24 and 16 x 26 bits
2K bytes, 2-way	2 x 128 x 64 bits	32 x 23 and 32 x 25 bits
4K bytes, 2-way	2 x 256 x 64 bits	64 x 22 and 64 x 24 bits
8K bytes, 2-way	2 x 512 x 64 bits	128 x 21 and 128 x 23 bits
16K bytes, 2-way	2 x 1,024 x 64 bits	256 x 20 and 256 x 22 bits
32K bytes, 2-way	2 x 2,048 x 64 bits	512 x 19 and 512 x 21 bits
64K bytes, 2-way	2 x 4,096 x 64 bits	1,024 x 18 and 1,024 x 20 bits
1K bytes, direct mapped	128 x 64 bits	32 x 23 bits
2K bytes, direct mapped	256 x 64 bits	64 x 22 bits
4K bytes, direct mapped	512 x 64 bits	128 x 21 bits
8K bytes, direct mapped	1,024 x 64 bits	256 x 20 bits
16K bytes, direct mapped	2,048 x 64 bits	512 x 19 bits
32K bytes, direct mapped	4,096 x 64 bits	1,024 x 18 bits
64K bytes, direct mapped	8,192 x 64 bits	2,048 x 17 bits

Table 22 lists the ICACHE signals that are connected to application specific RAMs. The IC_ prefix indicates signals that are driven by the ICACHE LMI module and received by the RAMs. The ICR_ prefix indicates signals that are driven by the ICACHE RAMs and received by the ICACHE LMI. Lexra supplies the Verilog module that makes all required connections to these wires. The width of the index and data lines depends upon the RAM connected to the LMI, and can be inferred from the Table 21.

Table 22: ICACHE RAM Interfaces

Signal	Description
IC_TAGINDEX	Tag and state RAM address (line).
ICR_TAGRD0	Tag and state RAM element 0 read path.
IC_TAGWR0	Tag and state RAM element 0 write path.
ICR_TAGRD1	Tag and state RAM element 1 read path.
IC_TAGWR1	Tag and state RAM element 1 write path.
IC_TAG0WE<N>	Tag 0 RAM write enable.
IC_TAG0RE<N>	Tag 0 RAM read enable.
IC_TAG0CS<N>	Tag 0 RAM chip select.

Signal	Description
IC_TAG1WE<N>	Tag 1 RAM write enable.
IC_TAG1RE<N>	Tag 1 RAM read enable.
IC_TAG1CS<N>	Tag 1 RAM chip select.
IC_INSTINDEX	Instruction RAM address (word).
ICR_INST0RD	Instruction RAM element 0 read path.
ICR_INST1RD	Instruction RAM element 1 read path.
IC_INSTWR	Instruction RAM write path (to both ways).
IC_INST0WE<N>[1:0]	Instruction RAM 0 write enable.
IC_INST0RE<N>	Instruction RAM 0 read enable.
IC_INST0CS<N>	Instruction RAM 0 chip select.
IC_INST1WE<N>[1:0]	Instruction RAM 1 write enable.
IC_INST1RE<N>	Instruction RAM 1 read enable.
IC_INST1CS<N>	Instruction RAM 1 chip select.

Note: <N> designates an available active-low version of a signal.

7.5. Instruction Memory (IMEM) LMI

The IMEM LMI supplies the interface for an optional local instruction store. The IMEM serves a fixed range of the physical address space, determined by configuration settings in *lconfig*. The IMEM contents are filled and invalidated under the control of the CP0 CCTL register, described in Section 7.2, Cache Control Register: CCTL. The IMEM module services instruction fetches that falls within its configured range. The IMEM is a convenient, low-cost alternative to a cache that makes instruction memory available to the core for high-speed access.

The configurations supported by IMEM, and the synchronous RAMs required for each, are summarized in Table 23.

Table 23: IMEM Configurations

Configuration	IMEM_INST RAM
no local instruction RAM	no RAM required
1K bytes	128 x 64 bits
2K bytes	256 x 64 bits
4K bytes	512 x 64 bits
8K bytes	1,024 x 64 bits
16K bytes	2,048 x 64 bits
32K bytes	4,096 x 64 bits

Configuration	IMEM_INST RAM
64K bytes	8,192 x 64 bits
128K bytes	16,384 x 64 bits
256K bytes	32,768 x 64 bits

Table 24 lists the IMEM signals that are connected to application specific RAMs. The *IW_* prefix indicates signals that are driven by the IMEM LMI module and received by RAMs. The *IWR_* prefix indicates signals that are driven by RAMs and received by the IMEM LMI. The *CFG_* prefix identifies configuration ports on the IMEM LMI that are typically wired to constant values. The width of the index and data lines depends upon the RAM connected to the LMI, and can be inferred from Table 23.

The *CFG_* wires define where the IMEM is mapped into the physical address space. This configuration information defines the local bus address region of the IMEM. It also determines the main memory locations that are accessed by the LX4380 when an IMEM fill operation is started (by updating the IMEMFill bit of the CP0 CCTL register). The *lconfig* utility supplied by Lexra will verify that the configured address range does not interfere with other regions defined for LX4380. The size of the memory region must be a power of two, and must be naturally aligned.

Table 24: IMEM RAM Interfaces

Signal	Description
IW_INSTINDEX	IMEM index.
IWR_INSTRD	Instruction read data.
IW_INSTWR	Instruction write data.
IW_INSTWE<N>[1:0]	Instruction RAM write enable.
IW_INSTRE<N>	Instruction RAM read enable.
IW_INSTCS<N>	Instruction RAM chip select.
CFG_IWBASE[31:10]	Configured base address (modulo 1K bytes).
CFG_IWTOP[17:10]	Configured top address (bits that may differ from base).

Note: <N> designates an available active-low version of a signal.

7.6. Data Cache (DCACHE) LMI

The DCACHE LMI supplies the interface for a data cache attached to the LX4380 local bus. The data cache is RTL configurable for direct mapped or two-way set associativity, and write-back or write-through operation. The data cache participates in cacheable data reads and writes, but only if the address is not claimed by the DMEM LMI. The configurations supported by the data cache and the synchronous RAMs required for each are summarized in Table 25.

See Section D.4, Load/Store Rules, for detailed descriptions of pipeline stalls that the data cache may cause.

Writes that miss the cache or writes that are performed in write-through mode may require extra time to be serviced by the LBC if its write buffer is full.

Table 25: DCACHE Configurations

Configuration	DCACHE_DATA RAM	DCACHE_TAG RAM
no data cache	no RAM required	no RAM required
1K bytes, 2-way	2 x 64 x 64 bits	16 x 24 and 16 x 26 bits
2K bytes, 2-way	2 x 128 x 64 bits	32 x 23 and 32 x 25 bits
4K bytes, 2-way	2 x 256 x 64 bits	64 x 22 and 64 x 24 bits
8K bytes, 2-way	2 x 512 x 64 bits	128 x 21 and 128 x 23 bits
16K bytes, 2-way	2 x 1,024 x 64 bits	256 x 20 and 256 x 22 bits
32K bytes, 2-way	2 x 2,048 x 64 bits	512 x 19 and 512 x 21 bits
64K bytes, 2-way	2 x 4,096 x 64 bits	1,024 x 18 and 1,024 x 20 bits
1K bytes, direct mapped	128 x 64 bits	32 x 23 bits
2K bytes, direct mapped	256 x 64 bits	64 x 22 bits
4K bytes, direct mapped	512 x 64 bits	128 x 21 bits
8K bytes, direct mapped	1,024 x 64 bits	256 x 20 bits
16K bytes, direct mapped	2,048 x 64 bits	512 x 19 bits
32K bytes, direct mapped	4,096 x 64 bits	1,024 x 18 bits
64K bytes, direct mapped	8,192 x 64 bits	2,048 x 17 bits

Table 26 lists the DCACHE signals that are connected to application specific RAMs. The DC_ prefix indicates signals that are driven by the DCACHE LMI module and received by the RAMs. The DCR_ prefix indicates signals that are driven by the DCACHE RAMs and received by the DCACHE LMI. Lexra supplies the Verilog module that makes all required connections to these wires. The width of the index and data lines depends upon the RAM connected to the LMI, and can be inferred from Table 25.

Table 26: DCACHE RAM Interfaces

Signal	Description
DC_TAGINDEX	Tag and state RAM address.
DCR_TAGRD	Tag and state RAM read path.
DC_TAGWR	Tag and state RAM write path.
DC_TAGWE<N>	Tag and state RAM write enable.
DC_TAGRE<N>	Tag and state RAM read enable.
DC_TAGCS<N>	Tag and state RAM chip select.
DC_DATAINDEX	Data RAM address (word).
DCR_DATARD	Data RAM read path.
DC_DATAWR	Data RAM write path.

Signal	Description
DC_DATAWE<N>[1:0]	Data RAM write enable.
DC_DATARE<N>	Data RAM read enable.
DC_DATACS<N>	Data RAM chip select.

Note: <N> designates an available active-low version of a signal.

When configured for write-back operation, the data cache tag RAM includes a bit to indicate that a line is dirty. Each cache line is covered by a single dirty bit which when set indicates that the processor has modified the line in the cache but has not updated main memory. When a line is filled from system memory, the dirty bit is cleared. If a write hits in the cache and the dirty bit is not set (a clean line), the data cache RAM is updated with the write data and the dirty bit is set to one. If the line is already dirty when a write hits in the cache, the data cache RAM is updated with the write data and the dirty bit remains set. Any cached write that hits the write-back data cache updates the cache only, and does not cause any system bus activity.

When configured as a write-back cache, the data cache LMI also includes an evict buffer. In the case of a read miss to a dirty line, the data cache first issues a line read operation to fetch the new line. If the line currently stored in the cache is dirty, the line is copied from the data cache RAM to the evict buffer. When the current line has been completely copied into the evict buffer, the new line is loaded into the data cache RAM. As soon as the evict buffer is full, the data cache issues a line write operation. The processor does not stall while the line is being written, unless the processor causes the data cache to issue another system bus operation before the line write operation is complete.

Cache lines are only allocated on read misses, not writes. If a write misses in the cache, it will be issued as a single write on the bus and no line will be evicted or filled. This is the same for both write-back and write-through caches.

The replacement policy for the 2-way set-associative configuration is LRU (Least Recently Used).

Table 27 shows the data cache and system bus activity based on the current operation, the state of the line currently stored at the cache location and the outcome of the tag compare. The table includes some unusual cases, such as a uncached operation hitting the data cache. Such conditions are possible because the same physical address can be accessed in both cacheable or uncacheable modes, either through a kseg0/kseg1 address alias, or through mappings that are in effect with the optional MMU. The data cache controller treats these cases in a conservative fashion to ensure coherency between the data cache and main memory.

Table 27: Data Cache Operations and Results

Operation			State of Line Currently Stored in Cache	Tag Compare Result	Action	New Cache State
Cmd	Cached/ Uncached	Write-Through/ Write-Back				
Read	Cached	X	Invalid	X	Issue a line fill	Clean
		X	Clean	Hit	Read from cache	Clean
		X	Clean	Miss	Invalidate and issue line fill	Clean
		X	Dirty	Hit	Read from cache	Dirty
		X	Dirty	Miss	Evict line and issue line fill	Clean
	Uncached	X	Invalid	X	Read from system bus	Invalid
		X	Clean	Hit	Invalidate and read from system bus	Invalid
		X	Clean	Miss	Read from system bus	Clean
		X	Dirty	Hit	Evict line and read from system bus	Invalid
		X	Dirty	Miss	Read from system bus	Dirty
Write	Cached	X	Invalid	X	Write to system bus	Invalid
		write-back	Clean	Hit	Write to cache only	Dirty
		write-through	Clean	Hit	Write to cache and system bus	Clean
		X	Clean	Miss	Write to system bus	Clean
		write-back	Dirty	Hit	Write to cache only	Dirty
		write-through	Dirty	Hit	Write to cache and system bus	Dirty
		X	Dirty	Miss	Write to system bus	Dirty
	Uncached	X	Invalid	X	Write to system bus	Invalid
		X	Clean	Hit	Invalidate and write to system bus	Invalid
		X	Clean	Miss	Write to system bus	Clean
		X	Dirty	Hit	Evict line and write to system bus	Invalid
		X	Dirty	Miss	Write to system bus	Dirty

X = don't care

7.7. Scratch Pad Data Memory (DMEM) LMI

The DMEM LMI supplies the interface for a scratch pad data RAM attached to the LX4380 local bus. The DMEM module services any cacheable or uncacheable data read or write operation that falls within its configured range.

DMEM can perform reads or writes that hit DMEM at the rate of one per cycle. See Section D.4, Load/Store Rules, for detailed descriptions of pipeline stall conditions that may be caused by DMEM.

Because a write operation to the DMEM is never sent to the system bus, writes to DMEM will not cause processor stalls because of pending system bus activity.

The DMEM configurations and the synchronous RAMs required for each are summarized in the Table 28.

Table 28: DMEM Configurations

Configuration	DMEM_DATA RAM (64-bit)	DMEM_DATA RAM (128-bit)
no local data memory	no RAM required	no RAM required
1K bytes	128 x 64 bits	64 x 128 bits
2K bytes	256 x 64 bits	128 x 128 bits
4K bytes	512 x 64 bits	256 x 128 bits
8K bytes	1,024 x 64 bits	512 x 128 bits
16K bytes	2,048 x 64 bits	1,024 x 128 bits
32K bytes	4,096 x 64 bits	2,048 x 128 bits
64K bytes	8,192 x 64 bits	4,096 x 128 bits
128K bytes	16,384 x 64 bits	8,192 x 128 bits
256K bytes	32,768 x 64 bits	16,384 x 128 bits

Table 29 lists the DMEM signals that are connected to application specific RAMs. The *DW_* prefix indicates signals that are driven by the DMEM LMI module and received by RAMs. The *DWR_* prefix indicates signals that are driven by RAMs and received by the DMEM LMI. The *CFG_* prefix identifies configuration ports on the DMEM LMI that are typically wired to constant values. The width of the index and data lines depends upon the RAM connected to the LMI, and can be inferred from Table 28.

The *CFG_* wires define where DMEM is mapped into the physical address space. It is not possible for any DMEM reference to result in an operation on the system bus. The *lconfig* utility supplied by Lexra will verify that the configured address range does not interfere with other regions defined for LX4380. The size of the memory region must be a power of two, and must be naturally aligned.

Table 29: DMEM RAM Interfaces

Signal	Description
DW_DATAINDEX	Decoded data RAM index.
DWR_DATARD	Data RAM read data.
DW_DATAWR	Data RAM write data.

Signal	Description
DW_DATAWE<N>	Data RAM write enable.
DW_DATARE<N>	Data RAM read enable
DW_DATACS<N>	Data RAM chip select
CFG_DWBASE[31:10]	Configured base address (modulo 1K bytes).
CFG_DWTOP[17:10]	Configured top address (bits that may differ from base).

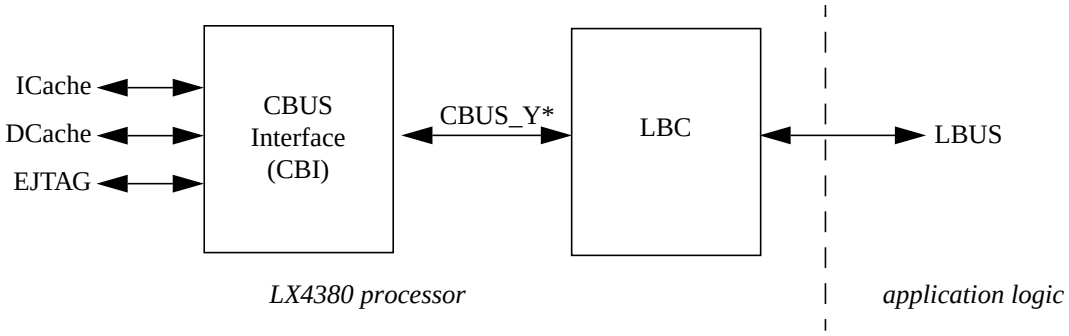
Note: <N> designates an available active-low version of a signal.

8. CBUS Interface

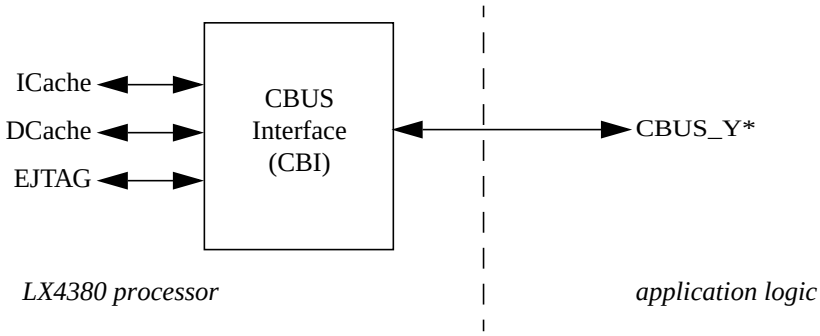
This section describes the CBUS, a system interface to the LX4380 that is an alternative to the LBUS. The CBUS Interface (CBI) provides a simple signalling layer between the LX4380 processor's cache controllers and the optional LX4380 system bus interface, the LBC. (See Section 9 for information on the LBC and LBUS.) LX4380 applications that connect to a bus protocol other than LBUS may eliminate the LBC and provide their own system bus interfaces or devices that connect directly to the LX4380 using CBUS.

8.1. System Interface Configuration

Figure 8 illustrates the LX4380's organization for the different system interface configurations. This section describes the configuration shown is part (b) of the figure.



(a) LX4380 Configured with LBUS Interface



(b) LX4380 Configured with CBUS Interface

Figure 8: LX4380 System Interface Configurations

8.2. CBUS Interface Write Buffer and Out-of-Order Processing

The CBUS Interface contains a write buffer with a depth that is configurable with *lconfig*. All write requests from the CPU are posted in the write buffer. The CPU will not wait for the write to complete. Write operations complete in the order they are entered into the buffer. If the buffer is full and the data cache generates another write operation to the CBUS Interface, then the CPU is stalled until an entry becomes available. LX4380 applications that employ LBUS instead of CBUS still use the CBI write buffer.

When the CPU issues a read operation, the CBI will attempt to forward that request to the Lexra Bus ahead of any pending write operations. This significantly improves performance since the CPU must wait for the read operation to complete.

There are a few cases when the CBI will not allow the read operation to pass pending writes:

1. The address of a pending write is within the same cache line as a data cache read. The CBI will hold the read operation until the matching write operation, and all write operations ahead of it, complete. If the read is for an instruction fetch, it can still pass a pending write that is inside the same cache line.

2. A data cache read is to uncacheable address space. All writes complete before the read is issued. This avoids any problems with I/O devices and their associated control/status registers.

3. A pending write is to uncachable address space. The CBI holds the read operation until all writes up to and including the write to uncacheable address space complete. This further avoids I/O device problems.

The write buffer bypass feature can be disabled via *lconfig* so that reads never pass writes.

8.3. CBUS Line Read Interleave Order

The line read operation reads a sequence of data beats from memory corresponding to the size of a cache line. The cache line size affects how many cycles are required to transfer the entire line. The LX4380 supports a configurable line size, specified through *lconfig*. A line size of eight words (32 bytes) is assumed here.

The CBUS target may transfer the read data starting with *word zero first*, or starting with the *desired word first*. With word zero first operation, the target transfers four 64-bit beats of data in sequence, starting at the nearest 32-byte-aligned address smaller or equal to the address that the initiator drives. In other words, the target starts the transfer at the beginning of the line containing the requested address. With desired word first operation, the first data beat returned by the target is the beat corresponding to the address instead of the word zero of the line. The second beat is the next sequential data beat, and so on. At the end of the line, the target wraps around and returns the first beat of line. All devices attached to the CBUS must consistently return word zero first or the desired word first. The LX4380 is configurable to work with either mode.

The LX4380 supports two ways of incrementing the address of a line fill. One is by *linear wrap*, where the address is simply incremented by one. The other is by *interleaved wrap*, where the next address is determined by the logical xor of the cycle count and the first word address. The interleave sequence is shown in Table 30. The low-order address bits 4:3 for the first data beat are the obtained from the address of the line read request. The low-order address bits for the subsequent data indicate the corresponding interleave order. All devices attached to the CBUS must consistently support linear wrap or interleaved wrap. The LX4380 is configurable to work with either mode.

Table 30: Line Read Interleave Order

Interleaved	Address[4:3]			
1st data beat	00	01	10	11
2nd data beat	01	00	11	10
3rd data beat	10	11	00	01
4th data beat	11	10	01	00

8.4. CBUS Byte Alignment

CBUS data must be driven to the byte lanes according to the rules shown in Table 31. Alignments not shown are not generated by the CPU. All multi-beat operations transfer multiple twin word beats over CBUS.

Table 31: CBUS Byte Lane Assignment

		CBUS Bus data byte lanes used							
Transfer Size	**ADDR[2:0]**	**63:56**	**55:48**	**47:40**	**39:32**	**31:24**	**23:16**	**15:8**	**7:0**
byte	000	X							
byte	001		X						
byte	010			X					
byte	011				X				
byte	100					X			
byte	101						X		
byte	110							X	
byte	111								X
half word	000	X	X						
half word	010			X	X				
half word	100					X	X		
half word	110							X	X
word	000	X	X	X	X				
word	100					X	X	X	X
twin word	000	X	X	X	X	X	X	X	X

8.5. CBUS Interface Signal List

Table 32 summarizes the LX4380's CBUS signals.

Table 32: CBUS Signal List

Name	I/O	Function
CBUS_YREQO	O	0 - no request, 1 - processor is initiating a request
CBUS_YADDRO[31:0]	O	Address
CBUS_YREADO	O	1=Read, 0=Write
CBUS_YSZO[3:0]	O	Transfer size 4'b1000 - 1 byte 4'b1001 - 2 bytes 4'b1011 - 1 word 4'b1100 - 2 words 4'b0000 - 4 words This signal is don't care when CBUS_YLINEO is asserted.
CBUS_YLINEO	O	1 - line access, 0 - single access
CBUS_YDATAO[63:0]	O	Write Data
CBUS_YUCO	O	1 - uncached access, 0 - cached access
CBUS_YSRCO[3:0]	O	transaction source (within LX4380): 4'b0001 Instruction Cache 4'b0010 Data Cache or EJTAG DMA write 4'b0100 EJTAG DMA read 4'b1000 not used
CBUS_YDBUSYO	O	1 - LX4380 is not ready to receive Data. Any return with CBUS_YVALTYPEI of Data (4'b0010) is ignored by the LX4380. External logic must hold such data until CBUS_YDBUSYO is deasserted. 0 - LX4380 is ready to receive Data.
CBUS_YBUSYI	I	1 - External logic cannot accept request. The current CBUS_Y request, if any, is ignored by external logic. 0 - External logic is ready to accept a request.
CBUS_YDATAI[63:0]	I	Read Data
CBUS_YVALTYPEI[3:0]	I	Indicates valid read data of a certain type: 4'b0000 No valid read data 4'b0001 Instruction Cache 4'b0010 Data Cache 4'b0100 EJTAG DMA 4'b1000 not used
CBUS_YIDLEI	I	Indicates external CBUS_Y device is in an idle state, i.e. has no pending read or write transactions.

8.6. CBUS Transaction Types

The following transaction types are supported by the CBUS interface:

1. Single read.
2. Line read.
3. Single write.
4. Line writes.

8.7. CBUS Protocol

The transaction request protocol is controlled with CBUS_YREQO output and CBUS_YBUSYI input.

1. The CBUS_YREQO output is asserted by the LX4380 to initiate an access to external logic. Additional CBUS_Y* outputs are driven by the LX4380 to provide the transaction details.
2. CBUS_YREQO remains asserted until the CBUS_YBUSYI is not asserted by external logic.

For a write transaction, the transaction is completed after step 2. For a read transaction, additional steps control the return of read data by the external logic, using the CBUS_YVALTYPEI[3:0] input and the CBUS_YDBUSYO output.

1. If CBUS_YVALTYPEI indicates Instruction Cache or EJTAG DMA data is present on CBUS_YDATAI, the data is always accepted by the LX4380.
2. If CBUS_YVALTYPEI indicates that Data Cache data is present on CBUS_YDATAI and CBUS_YDBUSYO is asserted, the external logic must continue to drive CBUS_YVALTYPEI and CBUS_YDATAI until CBUS_YDBUSYO deasserts.

8.8. CBUS Transaction Timing Diagrams

Note: All of the following timing diagrams assume a line size of 8 words. For reads, the transaction request is shown in a different timing diagram than the returning read data as there is no protocol link between the two.

8.8.1. Back-to-Back Single Writes with Busy

In cycle 1 the write to address A is accepted by the external logic. In cycle 2 the external logic asserts CBUS_YBUSYI which causes the LX4380 to hold its request. In cycle 3, the external logic de-asserts CBUS_YBUSYI and accepts the request.

In this example, cycle 4 could be used by the processor to initiate another request.

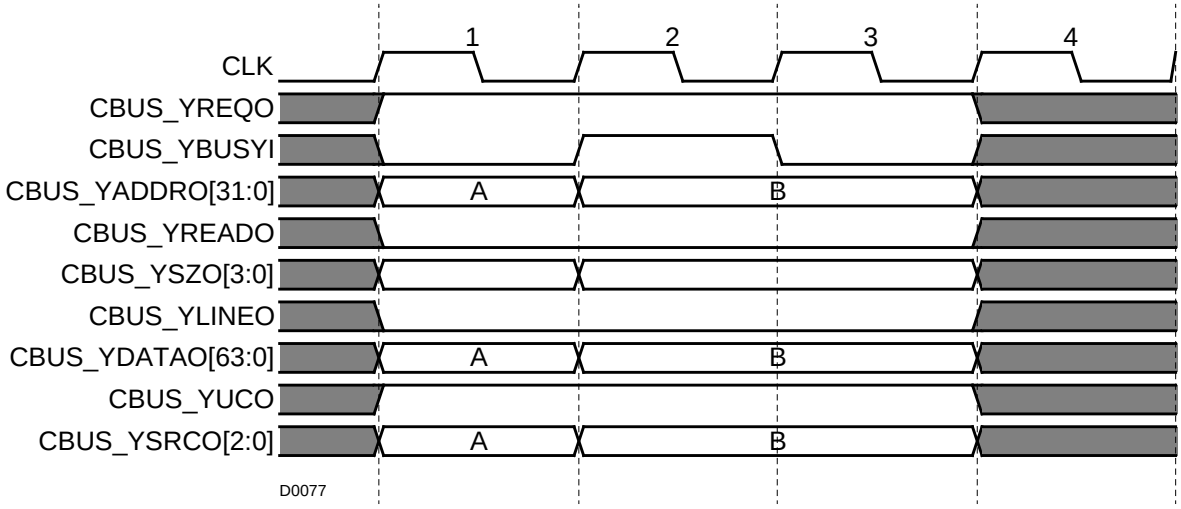


Figure 9: CBUS Back-to-Back Single Writes with Busy

8.8.2. Line Writes

During a line write the address is given in cycle 1. External logic signals that it is able to accept a line write request by de-asserting CBUS_YBUSYI. External logic does not honor a line write request when CBUS_YBUSYI is asserted.

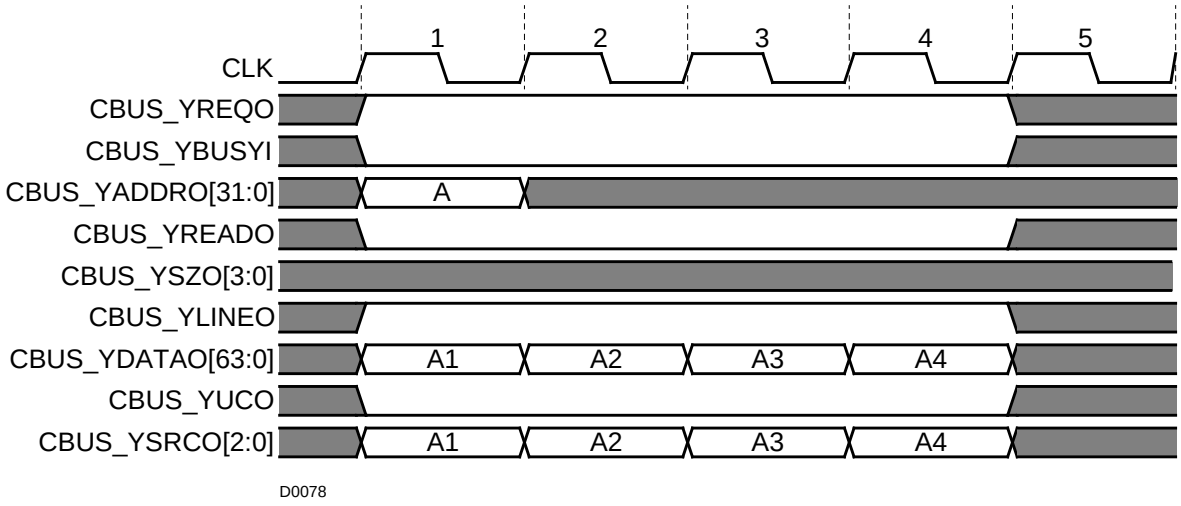


Figure 10: CBUS Line Write

8.8.3. Back-to-Back Single Read Requests with Busy

Only the read request is shown here. The return data is not shown.

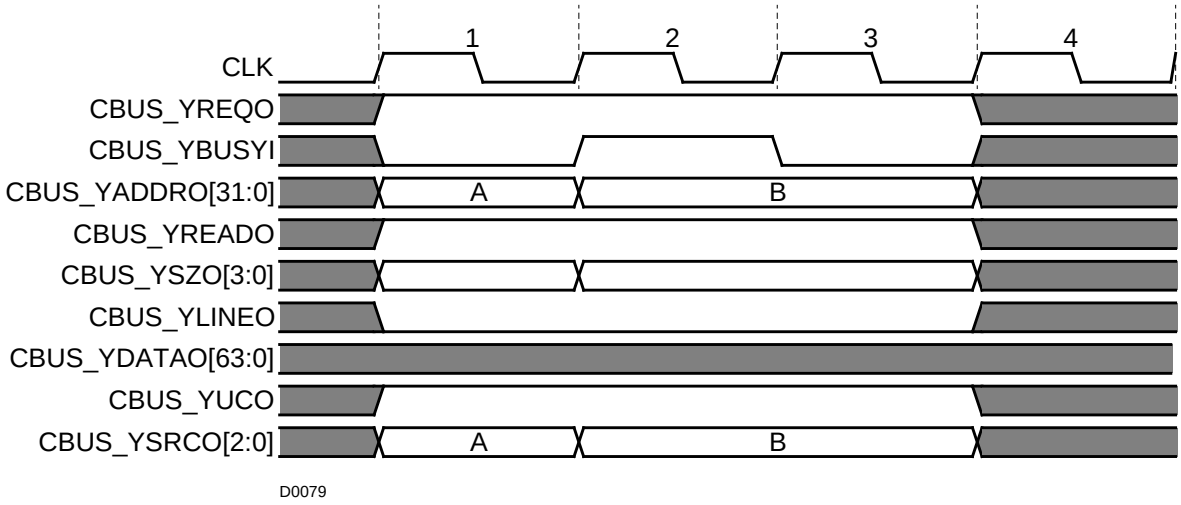


Figure 11: CBUS Back-to-Back Single Read Requests with Busy

8.8.4. Line Read Request

A line read request takes only one cycle with the data being returned later by the external logic.

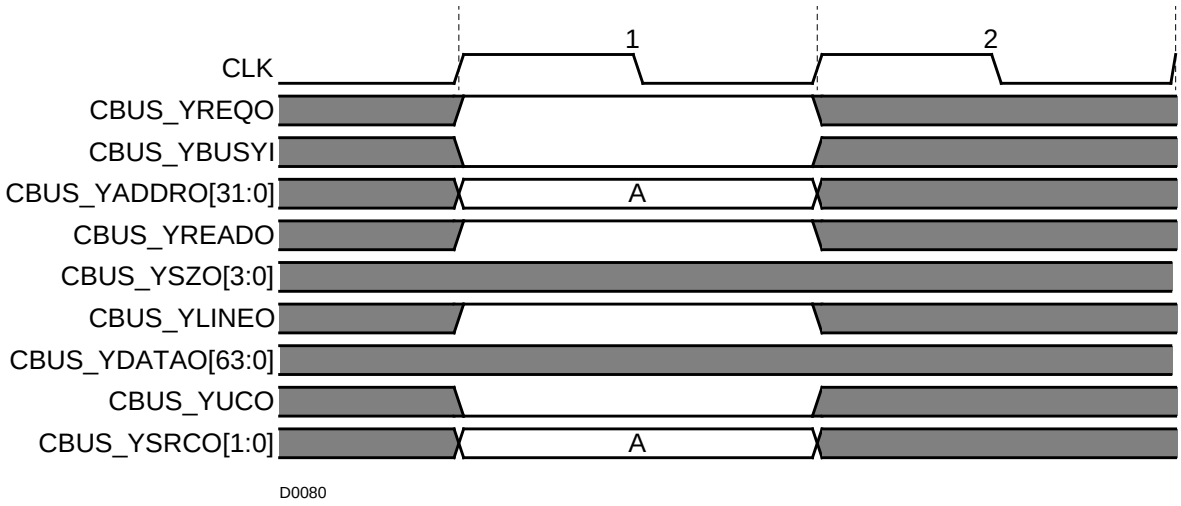


Figure 12: CBUS Line Read Request

8.8.5. Returning Read Data

External logic supplies read data on the CBUS_YDATAI inputs while asserting a bit within CBUS_YVALTYPEI. If CBUS_YVALTYPEI indicates Data (4'b0010), the LX4380 only accepts the read data if it has de-asserted CBUS_YDBUSYO. If CBUS_YDBUSYO is asserted, the external logic must maintain CBUS_YVALTYPEI and CBUS_YDATAI until CBUS_YDBUSYO is deasserted

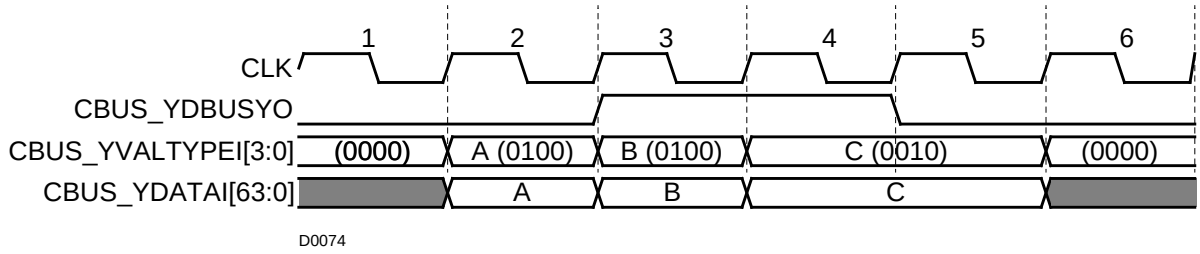


Figure 13: CBUS Read Data and DBUSY

A read line data return is illustrated below. The external device asserts the appropriate bit of CBUS_YVALTYPEI for each data beat. Assertion of CBUS_YDBUSYO is also illustrated.

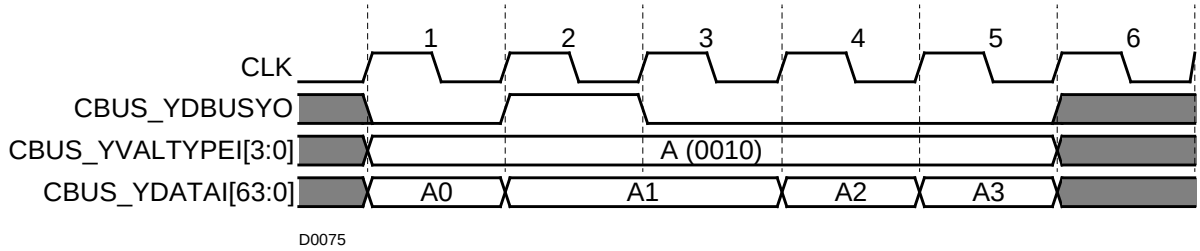


Figure 14: Read Data for a Line Read Request

8.8.6. Latency of CBUS Transactions

Figure 15 illustrates how the latency of a CBUS read transaction affects the duration of CPU stalls while the CPU waits for the read data to be returned.

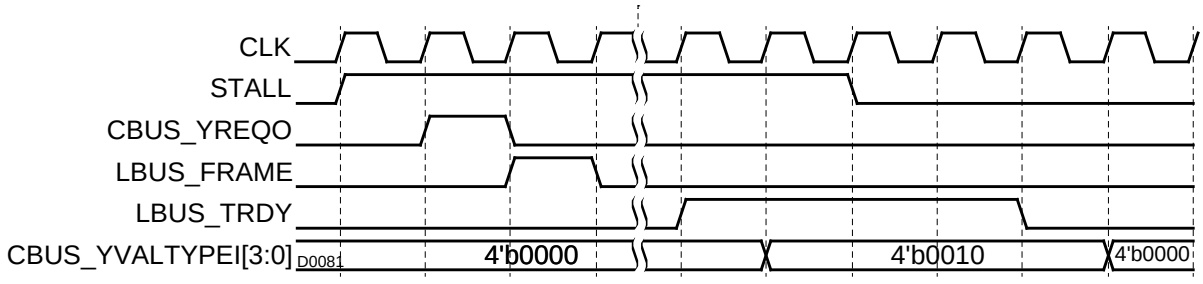


Figure 15: Latency of CBUS Transactions.

The overall latency encountered for any CBUS transaction depends more on system level behavior, and less on the behavior of the CBUS interface itself. In this example, the CBUS interface is synchronously connected to a full-speed LBUS via an LBC, and the LBC is assumed to be parked on the LBUS. Thus, the read transaction appears on LBUS one cycle after the CBUS request is initiated. Some number of cycles will pass as the addressed LBUS target prepares its data response. The LBUS target then supplies the data beats

coincident with the assertion of TRDY. The LBC requires only one cycle to pass the first data beat from the LBUS to the CBUS, at which time CBUS_YVALTYPE contains the code 5'b00010 to indicate the data response. The CBUS interface in turn requires only one cycle to pass the data to the CPU and release the stall condition.

From this example, it is seen that only three stall cycles can be attributed to the CBUS interface. If a synchronous full-speed LBUS is employed for the system bus, the LBC and LBUS protocol will result in a minimum of three additional stalls. The addressed LBUS target may also insert additional stalls.

9. Lexra System Bus (LBUS)

This section describes the optional Lexra System Bus (LBUS) and the Lexra Bus Controller (LBC) that connects the LX4380 to LBUS. The LBUS provides a flexible PCI-like protocol appropriate for systems with multiple masters and targets. Applications which do not require a such a system bus, or which include custom interfaces to other system buses, may optionally employ the LX4380's CBUS interface rather than the LBUS. (See Section 8, CBUS Interface.)

9.1. Connecting the LX4380 to Internal Devices

The Lexra Bus Controller (LBC) provides the connection between the LX4380 CBUS Interface (CBI) and the Lexra System Bus (LBUS). The LBUS supports application-specific system bus devices such as main memory, USB or IEEE-1394 (Firewire). The LBC uses a protocol similar to that of the Peripheral Component Interface (PCI) bus. This is a well-known and proven architecture. Adding new devices to the Lexra Bus is straightforward and the performance approaches the highest that can be achieved without adding a great deal of complexity to the protocol.

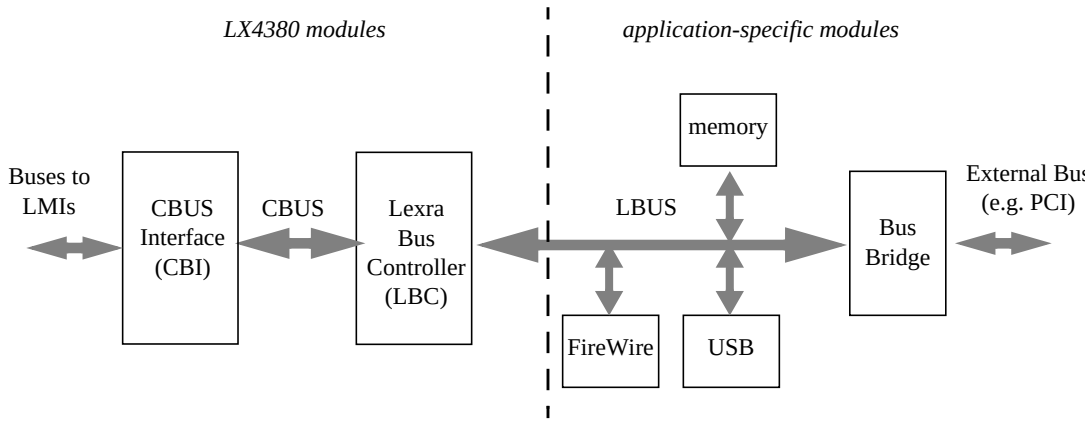


Figure 16: Lexra System Bus (LBUS) Diagram

The Lexra bus supports multiple masters. This allows for mastering I/O controllers with DMA engines to be connected to the bus. The bus has a pended architecture, in which a master holds the bus until all the data is transferred. This simplifies the design of user-supplied bus agents and reduces latency for cache miss servicing.

The Lexra bus is a synchronous bus. Signals are registered and sampled at the positive edge of the bus clock. Certain logical operations may be made to the sampled signals and then new signals can be driven immediately, such as for address decoding. This allows same-cycle turn-around. The LBC supports synchronous modes with the LBUS operating at full CPU speed or half CPU speed, and an asynchronous mode that allows the LBUS to be clocked at any speed independent of the CPU speed.

The Lexra bus data path for the LX4380 is 64 bits wide. Therefore, the bus can transfer two words, one word, a halfword, or a byte in one bus clock. The bus supports line and burst transfers in which several beats (64 bits) of data are transferred. The Lexra bus accomplishes this by transferring data beats in successive clock

cycles.

The LBC provides enabling signals to control application-specific muxes or tristate buffers. This allows the LBUS to have either a bi-directional or point-to-point topology.

9.2. Terminology

The Lexra bus borrows terminology from the PCI bus specification, on which the Lexra bus is partially based.

Bus transactions take place between two bus *agents*. One bus agent requests the bus and initiates a transfer. The second responds to the transfer.

The agent initiating a transfer is called the *bus initiator*. It is also referred to as the *bus master*. Both terms are used interchangeably in this document.

The responding agent is known as the bus *target*. It samples the address when it is valid, and determines if the address is within the domain of the device. If so, it indicates such to the initiator and becomes the target.

A *read transfer* is a bus operation whereby the master requests data from the target.

A *write transfer* is a bus operation whereby the master requests to send data to the target.

A *single data* bus operation is used to transfer two words, one word, a halfword, or a byte of data. The data can be transferred in one bus cycle, not including the address cycle and device latencies.

A *line transfer* is a read or write operation where an entire cache line of data is transferred in successive cycles as fast as the initiator and target can send/receive the data.

A *burst transfer* is a read or write operation where a large amount of data needs to be sent. The initiator presents a starting address and data is transferred starting at that address in successive cycles; for each word transferred, the address is incremented by the devices internally.

A *beat* is the data (up to 64 bits) that is transferred in one data cycle.

A *word* is 32 bits of data.

A device *asserts* a signal when it drives it to its logical true electrical state.

9.3. Bus Operations

The purpose of the Lexra bus is to connect together the various components of the system, including the LX4380 CPU, main system memory, I/O devices, and external bus bridges. Different devices have different transfer requirements. For example, the LX4380 CPU will request the bus to fetch a cache line of data from memory. I/O devices will request large blocks of data to be sent to and from memory. LBUS supports the various types of transfers needed by both I/O and the processor.

Single Data Read
Line Read
Burst Read
Single Data Write
Line Write
Burst Write

(Note that although burst operations are defined for the LBUS protocol, the LX4380 does not initiate burst operations.)

9.3.1. Single Data Read

The single data read operation reads a twinword, single word, halfword, or byte from the target device. This operation is usually used by the CPU to read data from uncachable address space. (If the read address was in cacheable address space, either a hit would occur resulting in no bus activity, or a miss would occur resulting in a read line transaction.)

9.3.2. Line Read

The line read operation reads a sequence of data beats from memory corresponding to the size of a cache line. The cache line size affects how many cycles are required to transfer the entire line. The LX4380 supports a configurable line size, specified through *lconfig*. A line size of eight words (32 bytes) is assumed here.

The target may transfer the read data starting with *word zero first*, or starting with the *desired word first*. With word zero first operation, the target transfers four 64-bit beats of data in sequence, starting at the nearest 32-bit aligned address smaller or equal to the address that the initiator drives. In other words, the target starts the transfer at the beginning of the line containing the requested address. With desired word first operation, the first data beat returned by the target is the beat corresponding to the address instead of the word zero of the line. The second beat is the next sequential data beat, and so on. At the end of the line, the target wraps around and returns the first beat of line. All devices on the system bus must operate consistently with respect to whether they return word zero first or the desired word first. The LX4380 is configurable to work with either mode of operation.

The LX4380 supports two ways of incrementing the address of a line read. One is *linear wrap*, where the address is simply incremented by one. The other is *interleaved wrap*, where the next address is determined by the logical xor of the cycle count and the first word address. The interleave sequence is shown in the table below. The low-order address bits 4:3 for the first data beat are the obtained from the address of the line read request. The low order address bits for the subsequent data indicate the corresponding interleave order. The address increment mode used for a line read operation is specified in the bus command, as described in Section 9.5. The LX4380 is configurable via *lconfig* to generate bus commands for either mode.

Table 33: Line Read Interleave Order

Interleaved	Address[4:3]			
1st data beat	00	01	10	11
2nd data beat	01	00	11	10
3rd data beat	10	11	00	01
4th data beat	11	10	01	00

9.3.3. Burst Read

The burst read operation transfers an arbitrary amount of data from the target to the initiator. The initiator first presents a starting address to the target. The target responds by providing multiple cycles of data beats in sequence, starting at the initial address. The initiator indicates to the target when to stop providing data.

Burst read operations are used by I/O devices for block DMA transfers. The LX4380 does not issue burst read operations.

Note that there is a difference between an 8-cycle burst and a line read. A line read may use a desired-word-first increment and wrap. A burst will always increment and will never wrap.

9.3.4. Single Data Write

The single data write operation writes a twinword, a single word, a halfword, or a byte to the target. (Although twinword writes are supported by the LBUS protocol, the LX4380 does not issue twinword writes.)

The LX4380 data cache is configurable for write-through or write-back policies. CPU data writes that are performed in write-through mode generate a single data write operation on the system bus. CPU data writes that miss the data cache, even in write-back mode, also generate a single cycle write operation. However, the data cache inhibits these bus write operations if the address falls within the CPU's local DMEM.

9.3.5. Line Write

The line write operation write a sequence of data from memory corresponding to the size of a cache line. The cache line size affects how many cycles are required to transfer the full line. The LX4380 and the Lexra bus support a configurable line size, specified through *lconfig*. A line size of eight words (32 bytes) is assumed here. Line writes always begin with word zero as the first data beat.

9.3.6. Burst Write

A burst write is an operation where the initiator sends an address and then an indefinite sequence of data to the target. The initiator will inform the target when it has finished sending data. This operation is used by I/O devices for DMA transfers. It is not used by the LX4380.

9.4. Signal Descriptions

Table 34: LBUS Signal Description

Signal Name	Source (Initiator/Target/Ctrl)	Description
BCLOCK	Ctrl	Bus Clock
BCMD[6:0]	Initiator	Encoded command. Active during first cycle that BFRAME is asserted.
BADDR[31:0]	Initiator	Address; Target indicates valid address by asserting BFRAME.
BFRAME	Initiator	Asserted by initiator at beginning of operation with address and command signals; de-asserted when initiator is ready to accept or send last piece of data. Other bus masters sample this and BIRDY to indicate that the bus will be available on the next cycle.
BIRDY	Initiator	For writes, indicates that initiator is driving valid data; on reads, indicates that initiator is ready to accept data.
BDATA[63:0]	Initiator on write/Target on read	Data; if driven by initiator, BIRDY indicates valid data on bus; if driven by target, BTRDY indicates valid data on bus.
BTRDY	Target	For writes, indicates that target is ready to accept data; on reads, indicates that target is driving valid data.
BSEL	Target	Asserted by selected target after initiator asserts BFRAME; indicates that target has decoded address and will respond to the transaction (i.e. has been selected).

9.5. LBUS Commands

The initiator drives BCMD during the cycle that BFRAME is asserted. The encoding for BCMD is shown below.

BCMD[6]	0	read
	1	write
BCMD[5:4]	00	burst, fixed length[a]
	01	burst, unlimited number of words
	10	line, interleaved wrap[b]
	11	line, linear wrap
BCMD[3:0]	1000	1 byte
	1001	2 bytes
	1010	reserved
	1011	1 word
	1100	2 words
	1101	reserved
	111x	reserved
	0000	4 words
	0001	8 words
	0010	16 words
	0011	32 words
	01xx	reserved

a. For burst transfers, the length is determined BCMD[3:0]
b. For line transfers the length is determined by the RTL line size configuration (set with *lconfig)*, not BCMD[3:0]

9.6. LBUS Byte Alignment

LBUS data must be driven to the byte lanes according to the rules shown in Table 35. Alignments not shown are not legal. All multi-beat operations transfer multiple twin word beats over LBUS.

Table 35: LBUS Byte Lane Assignment

		Lexra Bus data byte lanes used							
BCMD[3:0]	**ADDR[2:0]**	**63:56**	**55:48**	**47:40**	**39:32**	**31:24**	**23:16**	**15:8**	**7:0**
1000	000	X							
1000	001		X						
1000	010			X					
1000	011				X				
1000	100					X			
1000	101						X		
1000	110							X	
1000	111								X
1001	000	X	X						
1001	010			X	X				
1001	100					X	X		
1001	110							X	X
1011	000	X	X	X	X				
1011	100					X	X	X	X
1100	100	X	X	X	X	X	X	X	X

The Lexra Bus does not define unaligned data transfers, such as a halfword transfer that starts at ADDR[1:0]=01, or transfers that would need to wrap to the next data beat.

9.7. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the element of the LX4380 that connects to the Lexra Bus. It forwards all transaction requests from the LX4380 CPU to the Lexra Bus. It is an initiator and will never respond to requests from other Lexra Bus initiators.

9.7.1. LBC Commands

The LBC issues only the LBUS commands listed in the table below.

Table 36: LBUS Commands Issued by the LBC

Command	BCMD[6]	BCMD[5:4]	BCMD[3:0]	Circumstances
Read Line	0	10 or 11, depending on configu-ration	undefined	A cache miss during a read by the CPU.
Read Single (twinword/word/half-word/byte)	0	00	10xx or 1100	A read by the CPU from an address in uncachable address space.
Write Line	1	10 or 11, depending on configu-ration	xxxx	When the data cache is configured for write-back operation, a read miss requires replacement of a dirty line.
Write Single (word/halfword/byte)	1	00	10xx	A write by the CPU into cacheable or uncachable address space.

9.7.2. Write Buffer

The LX4380 includes a write buffer in its CBUS Interface. When the LX4380 is configured to include the LBC, the CBUS Interface and its write buffer are always included as an internal LX4380 module. See Section 8.2, for a description of the write buffer.

9.7.3. LBC Read Buffer

The LBC contains a read buffer with a depth that is configurable with *lconfig*. All incoming read data from the system bus passes through the read buffer. This allows the LBC to accept incoming data as a result of a cache line fill operation without having to hold the bus.

When the LBC is configured with an asynchronous interface, a larger read buffer improves system and processor performance in the event of a cache miss. When the LBC is configured with a synchronous interface, the cache can accept data as fast as the LBC can transfer it. There is no need for a large read buffer. Through *lconfig*, the size of the read buffer may be reduced to a minimum size of two 64-bit data entries.

In some applications, there is a need to minimize the number of gates. The read buffer size may be reduced to two entries for the asynchronous case. This causes a penalty in terms of LBUS utilization since the LBC may have to delay the read (by de-asserting IRDY) if it cannot hold part of the line of data. When the read buffer is the size of a cache line, this will be rare since simultaneous instruction cache and data cache misses are relatively rare. For a smaller read buffer, delays are likely.

9.8. Transaction Descriptions

This section describes the various types of LBUS read and write transactions in detail. These operations adhere to the following protocols:

1. Agents that drive the bus do so as early as possible after the rising edge of the bus clock. There is some time to perform combinational logic after the bus clock goes high, but the amount of time is determined by the speed of the bus clock and the number of devices on the bus.

2. Agents sample signals on the bus at the rising edge of the bus clock.

3. All bus signals must be driven at all times. If the bus is not owned, and external device must drive the bus to a legal level.

4. A change in signal ownership requires one cycle during which the signal is not driven. If an initiator gives up the bus, another initiator needs to wait for one undriven cycle before it can drive the bus. If the same initiator issues a read operation and then needs to issue a write operation, it also must wait one extra cycle to ensure that the undriven cycle is present.

5. Agents that own signals must drive the signals to a logical true or logical false; all other agents must disable (tristate) their output buffers.

The Lexra Bus protocol is based on the PCI Bus protocol[1]. The Lexra Bus signals BFRAME, BTRY, BIRDY, and BSEL have a similar function to the PCI signals FRAME#, TRDY#, IRDY#, and DEVSEL#, respectively. In general, the protocol for the Lexra bus is as follows:

1. The initiator gains control of the bus through arbitration (described Section 9.10 on page 88).

2. During the first bus cycle of its ownership (before the first rising clock edge), the initiator drives the address for the bus transaction onto BADDR. At the same time, it asserts BFRAME to indicate that the bus is in use. It will de-assert BFRAME before it send or accepts the last data beat. In most cases, the initiator will assert BIRDY to indicate that it is ready to receive data (or read operations) or is driving valid data (for write operations). If the operation is a write, the initiator will drive valid data onto BDATA.

3. At the rising edge of the first clock, all agents sample BADDR and decode it to determine which agent will be the target.

4. The agent that determines that the address is within its address space asserts BSEL sometime after the first rising edge of the bus clock. BSEL stays asserted until the transaction is complete.

5. The initiator and the target transfer data either in one cycle or in successive cycles. The agent driving data (the initiator for a write, the target for a read) indicates valid data by asserting its ready signal (IRDY or TRDY for writes and reads, respectively). The agent receiving data (target for a write, initiator for a read) indicates its ability to receive the data by asserting its ready signal. Either agent may de-assert its ready signal to indicate that it cannot source or accept data on this particular clock edge.

6. When the initiator is ready to send or receive the last data beat, that is, when it asserts BIRDY for the last time, it also de-asserts BFRAME. It will de-assert BIRDY when the last data beat is transferred.

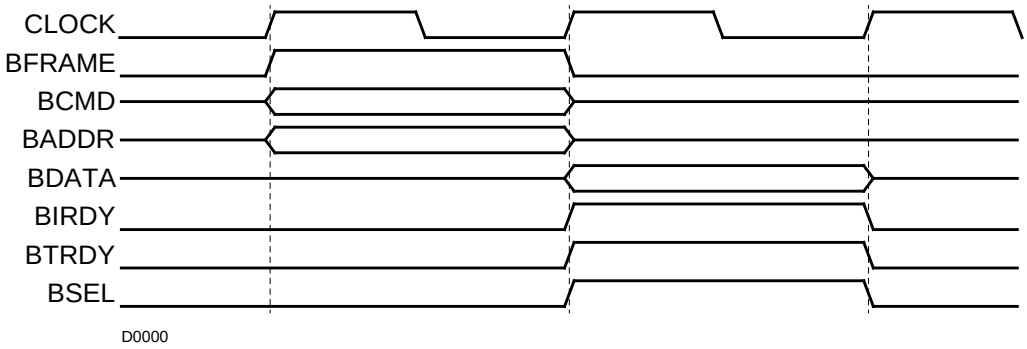
1. The Lexra Bus is not PCI compatible; it merely borrows concepts from the PCI Bus specification.

7. The arbiter grants the bus to the next initiator, and may do so during a bus transfer by a different initiator. The new initiator must sample BFRAME and BIRDY. When both BIRDY and BFRAME is sampled de-asserted and the new initiator has been given grant, it can assert BFRAME the next cycle to start a new transaction.

NOTE: in the examples below, the signals BADDR and BDATA are often shown to be in a high-impedance state. In reality, internal bus signals should always be driven, even if they are not being sampled. The Hi-Z states are shown for conceptual purposes only.

9.8.1. Single Data Read with No Waits

This operation is used to read a twinword, word, halfword or byte from memory, usually in uncachable address space.



This is a simple read operation where the target responds immediately with data. This is unlikely, since most devices will require one or more cycles to return data. This example illustrates the most basic read operation without waits.

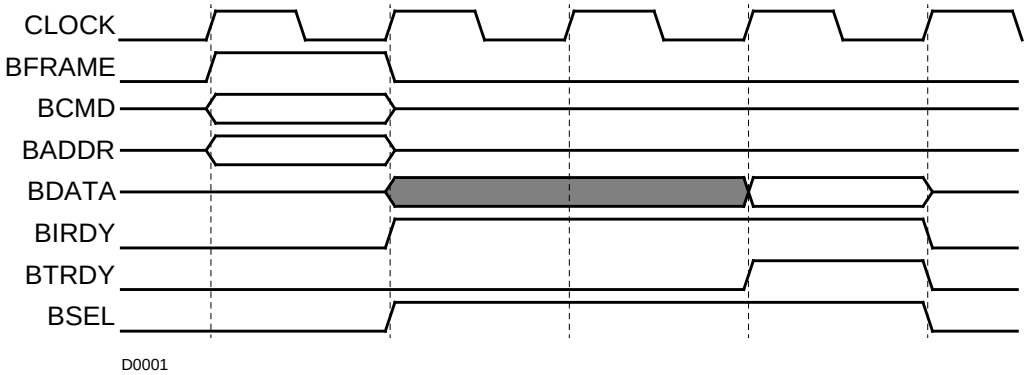
1. Initiator asserts BFRAME and drives BADDR.

2. Target asserts BSEL to indicate to initiator that a target is responding. In this example, there is an immediate fetch of data, so Target drives data and asserts BTRDY to indicate to target that it is driving data. The Initiator de-asserts BFRAME and asserts BIRDY to indicate that the next piece of data received will be the last.

3. Initiator de-asserts IBIRDY and the target de-asserts BSEL and BTRDY to indicate the end of the transaction. The Initiator that has been given grant owns the bus this cycle.

9.8.2. Single Data Read with Target Wait

This is the same as the single data read, except that the target needs time to fetch the data from memory.

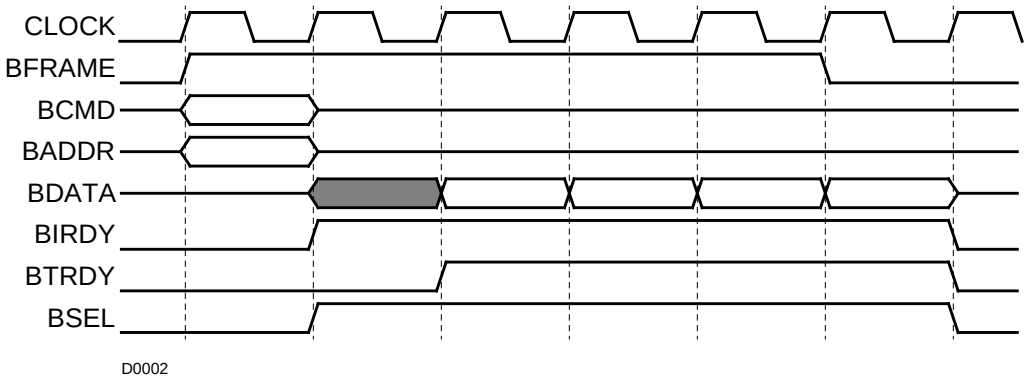


This is a common single data read operation.

1. Initiator asserts BFRAME and drives BADDR.
2. Target asserts BSEL to indicate that it has decoded the address and is acknowledging that it is the target device. However, it is not ready to send data, so it does not assert BTRDY. Initiator de-asserts BFRAME and asserts BIRDY to indicate that the next piece of data will be the last it wants.
3. Target has not asserted BTRDY so no data is transferred.
4. After a second wait cycle, target drives data and asserts BTRDY to indicate that data is on the bus.
5. Target de-asserts BSEL and BTRDY. Initiator de-asserts BIRDY. Another initiator may drive the bus this cycle.

9.8.3. Line Read with No Waits

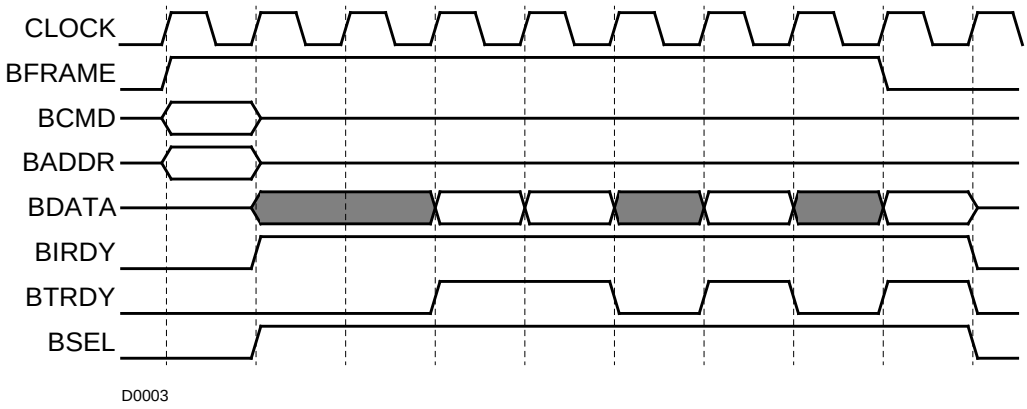
A Line Read transfers data beats that comprise a cache line. In this example, four data beats are transferred in sequence without any waits.



1. Initiator drives BADDR and asserts BFRAME to indicate beginning of transaction.
2. Target asserts BSEL to indicate that it had decoded the address and will send data when it is ready. Initiator asserts BIRDY to indicate that it is ready to receive data.
3. Target drives data and asserts BTRDY.
4. Target drives second data beat and continues to assert BTRDY.
5. Target drives third data beat and continues to assert BTRDY.
6. Target drives last data beat. Initiator de-asserts BFRAME to indicate that the next data beat it receives will be the last it needs.
7. Target de-asserts BTRDY and BSEL; initiator de-asserts BIRDY. Another master may gain ownership of the bus this cycle.

9.8.4. Line Read with Target Waits

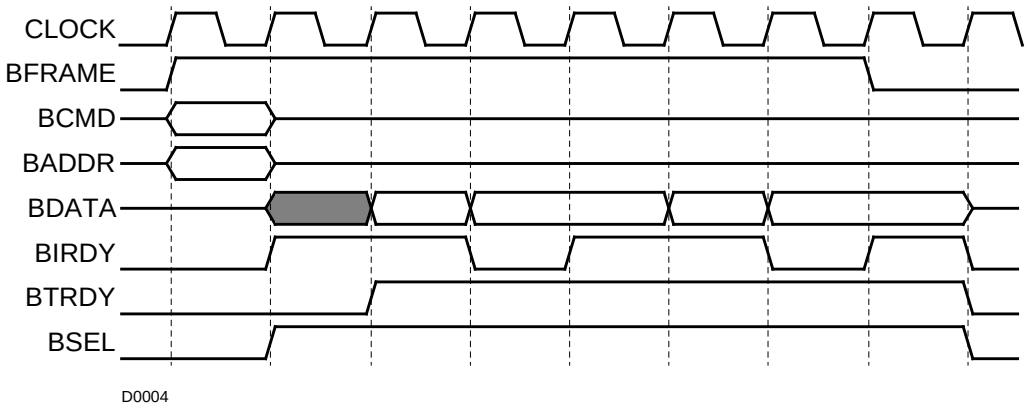
This illustrates what happens when a target needs extra time to fetch data it needs to service a cache miss.



1. Initiator asserts BFRAME and drives BADDR.
2. Target asserts BSEL to indicate that it is acknowledging the operation. Initiator asserts BIRDY to indicate that it is ready to receive data.
3. Target waits until it has the data.
4. Target drives first data beat and asserts BTDRY.
5. Target drives second data beat and asserts BTRDY.
6. Target cannot get third data beat, so it de-asserts BTRDY.
7. Target drives third data beat and asserts BTRDY.
8. Target cannot get fourth data beat, so it de-asserts BTRDY.
9. Target drives fourth data beat and asserts BTRDY.

9.8.5. Line Read with Initiator Waits

This occurs when a line of data is requested from the target and the initiator cannot accept all of the data in successive cycles.



1. Initiator drives address and asserts BFRAME.

2. Target asserts BSEL. It doesn't have data, so it does not assert BTRDY. Initiator asserts BIRDY to indicate that it can accept data

3. Target now has data, so it drives the data and asserts BTRDY.

4. Target drives second data beat; initiator cannot accept it, so it de-asserts BIRDY.

5. Target holds second data beat; initiator can accept it and asserts BIRDY.

6. Target drives third data beat; initiator accepts it.

7. Target drives fourth data beat; initiator cannot accept it and de-asserts BIRDY. initiator hold BFRAME until it can assert BIRDY.

8. Initiator asserts BIRDY to accept fourth data beat. It de-asserts BFRAME to indicate this is the last data beat.

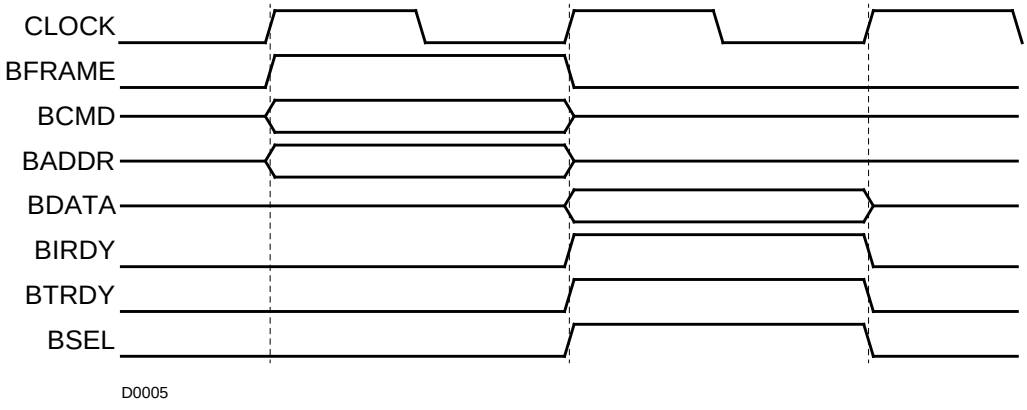
9.8.6. Burst Read

The burst read transaction is similar to a line read, except that BCMD indicates a burst read. The end of the burst is indicated when the initiator de-asserts BFRAME and BIRDY.

9.8.7. Single Data Write with No Waits

A single data write operation occurs when the LX4380 processor executes a store instruction that misses the data cache, or executes a store operation in write-through mode. Writes to uncacheable address space also generate a single data write. Single data write operations are used to write twinwords, words, halfwords and bytes. (But note, the LX4380 does not generate twinword writes.)

A single data write without waits requires two cycles.



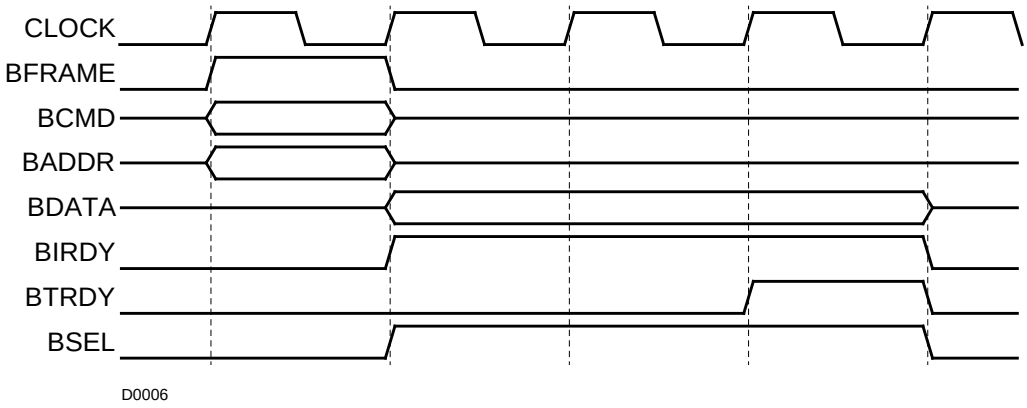
1. Initiator asserts BFRAME and drives address.

2. Target samples address and asserts BSEL. Initiator drives data and asserts BIRDY. In this case, target is also able to accept data, so it asserts BTRDY. Initiator also de-asserts BFRAME to indicate that it is ready to send the last (and only) data beat.

3. Target accepts data, de-asserts BTRDY and BSEL. Initiator de-asserts BIRDY.

9.8.8. Single Data Write with Waits

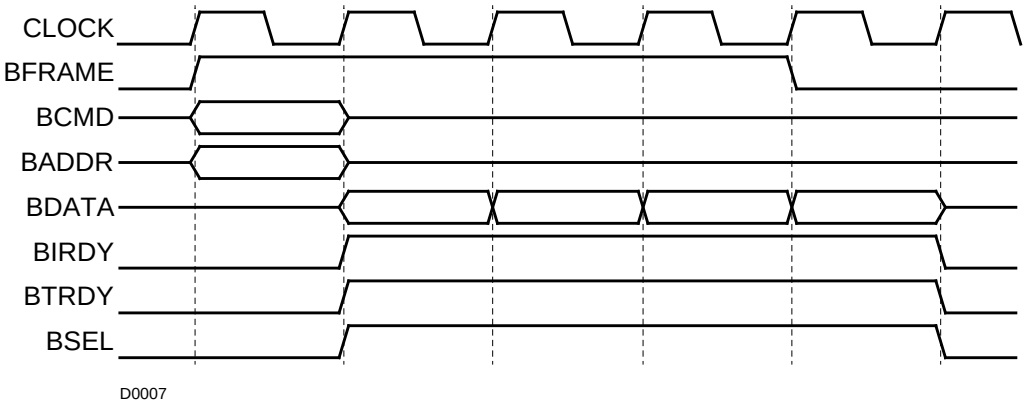
This is an example of a single data write operation where the target cannot immediately accept data and must insert wait states.



This is the same description as the above example, except that the target inserts two wait states until it asserts BIRDY to indicate acceptance of data.

9.8.9. Line Write with No Waits

A line write operation is generally used to transfer a modified cache line from a cache to main memory. The following illustrates a best-case scenario with no wait states.

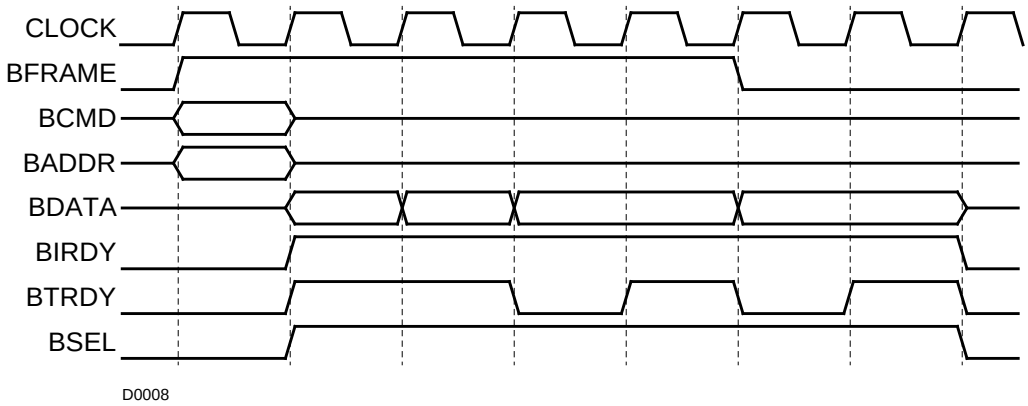


1. Initiator drives address and asserts BFRAME.
2. Target asserts BSEL and BTRDY to indicate it will accept data. Initiator drive data and asserts BIRDY.
3. Initiator drives next data beat; target continues to accept data and indicates as such by continuing to assert BTRDY.
4. Initiator drives third data beat; target continues to accept.
5. Initiator drives fourth data beat and de-asserts BFRAME to indicate that this will be its last beat sent; target accepts data.

6. Target de-asserts BTRDY and BSEL; initiator gives up control of the bus by de-asserting BIRDY.

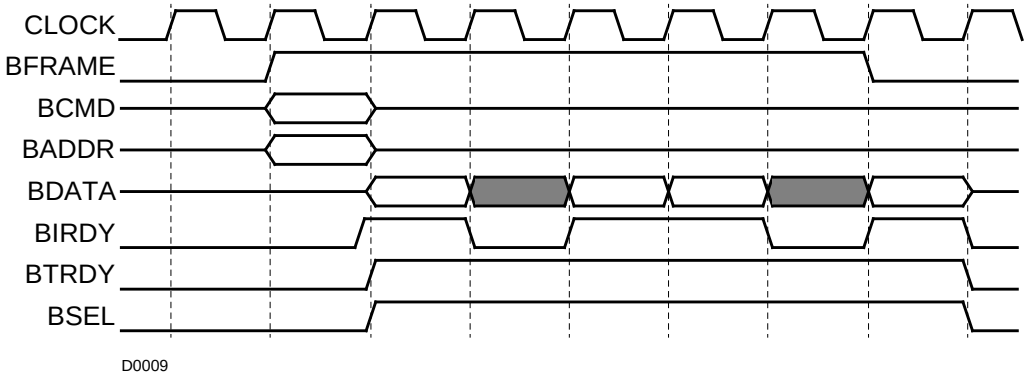
9.8.10. Line Write with Target Waits

This example is similar to the above example, except that during the third and fourth data beat transfer, the target cannot accept the data quickly enough, so it de-asserts BTRDY which indicates to the initiator that it should hold the data for an additional cycle.



9.8.11. Line Write with Initiator Waits

The example illustrates what happens when the initiator cannot supply data fast enough and has to insert waits.



9.8.12. Burst Write

A burst write is generally used to transfer large amounts of data from an I/O device to memory via a DMA transfer. This transaction is similar to a line write, except that BCMD indicates a burst write. The end of a burst write is indicated when the initiator de-asserts BFRAME and BIRDY.

9.9. LBC Signals

The table below summarizes the LX4380 LBC ports. The “LBC Port” column indicates the name of the port supplied by the LBC. The “Bus Signal” column indicates the corresponding Lexra bus signal. The LBC ports are strictly uni-directional, while the bus signals (at least conceptually) include multiple sources and sinks. The manner in which LBC ports are connected to bus signals is technology dependent, and may employ tri-state drivers or logic gating in conjunction with the LBC’s LCoe, LDoe and LToe outputs.

Table 37: LBC Interface Signals

I/O	LBC Port	Bus Signal	Description
output	LAddrO[31:0]	BADDR[31:0]	LBC address
output	LDataO[63:0]	BDATA[63:0]	LBC data
input	LDataI[63:0]	BDATA[63:0]	System data
output	LIrdy	BIRDY	LBC initiator ready
input	LIrdyI	BIRDY	System initiator ready
output	LFrame	BFRAME	LBC transaction frame
input	LFrameI	BFRAME	System transaction frame
input	LSelI	BSEL	System slave select
input	LTrdy	BTRDY	System target ready
output	LCmd[6:0]	BCMD[6:0]	LBC command
output	LReq	-	LBC bus request
input	LGnt	-	System bus grant
output	LCoe[9:0]	-	LBC command output enable terms
output	LDoe[7:0]	-	LBC data output enable terms
output	LToe	-	LBC transaction output enable terms

9.10. Arbitration

9.10.1. LBUS Rules

The following are the LBUS rules for arbitration.

REQ = a request from a master.

GNT = grant to the master.

idle = BFRAME and BIRDY are both de-asserted.

last = BIRDY and BTRDY are both asserted, and BFRAME is de-asserted.

busy = BIRDY or BTRDY or BFRAME are asserted.

1. Master asserts REQ at the beginning of a cycle and may start sampling for asserted GNT in the same cycle (in case GNT is already asserting in the case of a “park”).

2. If bus is *idle* or if the bus is in the *last* data phase of the previous transaction when master samples asserted GNT, then the master may drive BFRAME asserted on next cycle.

3. If the bus is *busy* when the master samples GNT, the master must also snoop BFRAME, BIRDY and BTRDY. If GNT is still asserted one cycle after BFRAME is de-asserted and both BIRDY and BTRDY are asserted (the last data phase), the master may drive BFRAME.

9.10.2. LBC Behavior

When the LBC needs access to LBUS, it asserts LReq and in the same cycle samples LGnt, ~LFrameI, and either ~LIrdyI or (LIRdyI & LTrdy). If these are true, the LBC takes ownership of the bus on the next cycle. The LBC de-asserts LReq the cycle after it asserts LFrame. If the bus is busy, the LBC continues to snoop these four signals for this condition. All other LBUS arbitration rules are based on this behavior of the LBC.

9.11. Connecting the LBC to LBUS

The LBC provides are three sets of output enables: LToe (valid for the length of the transaction), LCoe (valid for only the first cycle of a transaction), and LDoe (valid for data transfers, asserted by the master for writes and by the slave for reads).

LToe qualifies LFrame and LIrdy.

LCoe qualifies LCmd and LAddrO.

LDoe qualifies LDataO.

Application-specific devices may employ similar signals to qualify their LBUS outputs.

Instead of using the LBC's LToe and similar signals from application-specific bus devices, it may instead be desirable to logically OR the FRAME outputs from the LBC and all devices. This can be done either centrally or with one OR gate for each target and master. The same holds true for IRDY, TRDY, and SEL outputs. This simplifies the connections when a relatively few number of devices are used and there are no off-chip devices connected directly to the Lexra Bus.

Masters and slaves not taking part in a transaction must always keep their FRAME, IRDY, TRDY, and SEL outputs driven and de-asserted.

10. EJTAG Debug

Given the increasing complexity of SoC designs, the nature of embedded processor-design debug, hardware and software, and the time-to-market requirements of embedded systems, a debug solution is needed which allows on-chip processor visibility in a cost-effective, I/O constrained manner.

The EJTAG solution uses existing IEEE JTAG pins providing a method of debugging all devices accessible to the processor in the same way the processor would access those devices itself. Using EJTAG, a debug probe can access all the processor internal registers and caches. It can also access devices connected to the LX4380's CBUS or LBUS, bypassing internal caches and memories. SoC designers need only provide package connections to the LX4380's EJTAG signals to obtain the full benefits of embedded system debug using third party hardware probes and debug software.

EJTAG allows single-stepping through code and halting on breakpoints (hardware and software, address and data with masking). For debugging problems that are artifacts of real-time interactions, EJTAG gives real-time Program Counter (PC) trace capabilities from which an accurate program execution history is derived.

10.1. Overview

A debug host computer communicates to the EJTAG probe. The probe, in turn, communicates to the LX4380 EJTAG hardware via an IEEE 1149.1 JTAG interface. Through the use of the JTAG Test Access Port (TAP) controller, probe data is shifted into the EJTAG data and control registers in the LX4380 to respond to processor requests, DMA into system memory, configure the EJTAG control logic, enable single-step mode, or configure the EJTAG breakpoint registers. Through the use of the EJTAG control registers, the user can set hardware breakpoints on the instruction address, data address or data values.

Physical address range 0xFF20_0000 to 0xFF3F_FFFF is reserved for EJTAG use only and should not be mapped to any other device.

Currently, Embedded Performance Inc. (EPI) and Green Hills Inc. provide EJTAG debuggers and probes for the LX4380. Information on these products is available at the following web sites.

EPI Inc.: http://www.epitools.com

Green Hills Inc.: http://www.ghs.com

LX4380 EJTAG implements all required features of version 2.0.0 of the EJTAG specification, including:

- The LX4380 may access debug host resources via addressing of probe memory space.
- Debug host can DMA directly to or from devices attached to the LX4380's system bus.
- Hardware breakpoints may be installed on internal LX4380 instruction and data busses.
- EJTAG single-step execution mode.
- Real-time PC Trace.
- Debug exception and two EJTAG debug instructions: one for raising a debug exception via software, and one for returning from a debug exception.

10.1.1. IEEE JTAG-Specific Pinout

IEEE JTAG pins used by EJTAG are shown below. These are required for all EJTAG implementations. JTAG_TRST_N is an optional pin.

Table 38: EJTAG Pinout

Signal Name	I/O	Description
JTAG_TDO_NR	Output	Serial output of EJTAG TAP scan chain.
JTAG_TDI	Input	Serial input to EJTAG TAP scan chain.
JTAG_TMS	Input	Test Mode Select. Connected to each EJTAG TAP controller.
JTAG_CLOCK	Input	JTAG clock. Connected to each EJTAG TAP controller.
JTAG_TRST_N	Input	TAP controller reset. Connected to each EJTAG TAP controller.[a]

a. This pin is optional in multiprocessor configurations

Table 39: EJTAG AC Characteristics[1]

Signal	Parameter	Condition	Min	Max	Unit
JTAG_CLOCK	Frequency		<1	40	MHz
	Duty Cycle		40/60	60/40	%
JTAG_TMS	Setup to TCK rising edge	1.8V		5	ns
	Hold after TCK rising edge	1.8V		5	ns
JTAG_TDI	Setup to TCK rising edge	1.8V		5	ns
	Hold after TCK rising edge	1.8V		5	ns
JTAG_TDO_NR	Output Delay TCK falling edge to TDO	1.8V	0	7	ns

Table 40: EJTAG Synthesis Constraints[2]

Signal Name	Probe Budget	Core Budget	Slack remaining for other logic
JTAG_TDO_NR	0 to -7ns	11.5ns	13.5 to 20.5ns
JTAG_TDI	5ns	13.5ns	6.5ns
JTAG_TMS	5ns	13.5ns	6.5ns

1. Based on EPI Interface Specifications for MAJIC™ and MAJIC PLUS ™
2. Based on 25ns JTAG clock period.

10.2. Program Counter (PC) Trace

The LX4380 EJTAG includes support for real-time Program Counter (PC) Trace. When in PC Trace mode, the LX4380 serially outputs a new value of the program counter whenever there is a change in the PC (i.e. branch or jump instruction, or an exception).

When the PC Trace option is set to EXPORT in *lconfig*, the following signals will be output from the LX4380: DCLK, PCST, and TPC. These are described in more detail in the following subsections.

The DCLK output is used to synchronize the probe with the LX4380's core clock (SYSCLK).

The PCST (PC Trace Status) signals are used to indicate the status of program execution. Example status indications are sequential instruction, pipeline stall, branch, or exception.

The TPC pins output the value of the PC every time there is a change of program control.

10.2.1. PC Trace DCLK - Debug Clock

The maximum speed allowed for the Debug Clock (DCLK) output is 100MHz (as an EPI probe requirement). As cores typically run in excess of this speed DCLK can be set to a divided down value of SYSCLK. This is set by the DCLK N parameter in *lconfig*, which indicates the ratio of SYSCLK frequency to DCLK: 1, 2, 3 or 4.

10.2.2. PC Trace PCST - Program Counter Status Trace

The Program Counter Status (PCST) output comprises N sets of 3-bit PCST values, where N is the DCLK N parameter described in Section 10.2.1. A PCST value is generated every SYSCLK cycle. When DCLK is slower than the LX4380's SYSCLK, up to N PCST values are output simultaneously.

10.2.3. PC Trace TPC - Target Program Counter

The bus width of the Target Program Counter (TPC) output is user configured in *lconfig* via the "M" parameter to be one of 1, 2, 4 or 8 bits. When change in program flow occurs the current PC value is driven on the TPC output. As the PC is 32-bits wide, the number of TPC pins affects how quickly the PC is sent. For example, if the TPC is 4 bits wide the PC will take 8 DCLK cycles to be sent. If another change in flow occurs while the PC of the previous change is being transmitted, the new PC will be sent and the remainder of the previous PC will be lost unless the processor is in single-step mode. When an exception occurs, TPC also indicates the exception type with either 3 or 4 bits depending on whether or not vectored interrupts are present. This is described in more detail in Section 10.2.5.

The TDO output is used for the least significant bit of TPC (or the only bit if "M" is set to 1 via *lconfig*).

10.2.4. Single-Processor PC Trace Pinout

Table 41: Single-Processor PC Trace Pinout.

Signal Name	I/O	Description
JPT_TPC_DR M bits	O/P	The PC value is output on these pins when a PC-discontinuity occurs[a]
JPT_PCST_DR N*3 bits	O/P	PC Trace Status: Outputs current instruction type every DCLK
JPT_DCLK	O/P	PCST and TPC clock. Frequency determined as a fraction of SYSCLK via the N parameter. Maximum frequency of DCLK is 100MHz.

a. TPC[0] is multiplexed with TDO in the single-processor PC Trace solution.

Table 42: Single-Processor PC Trace AC Characteristics[1]

Signal	Parameter	Min	Max	Unit
JTAG_DCLK	Frequency	DC	100	MHz
DCLK	High Time	4		ns
	Low Time	4		ns
TPC	Setup to DCLK falling edge at probe	0		ns
	Hold after DCLK falling edge	4		ns
PCST	Setup to DCLK falling edge at probe	0		ns
	Hold after DCLK falling edge	4		ns

10.2.5. Vectored Interrupts and PC Trace

The EJTAG specification states that PC Trace provides a 3-bit code on the TPC output when an exception occurs (the PCST pins give the EXP code). In order to distinguish between the eight vectored interrupts in the LX4380 from all other exceptions, the LX4380 employs a 4-bit code.

For all exceptions other than vectored interrupts, the most significant bit of the 4-bit code is zero and the remaining 3-bits are the standard 3-bit code. Note that this includes the standard software and hardware interrupts numbered 0 through 7.

For vectored interrupts, the most significant bit is always 1. The 4-bit code is simply the number of the vectored interrupt (from 8 through 15) being taken.

Since the target of the vectored interrupt is determined by the contents of the INTVEC register, the debug software which monitors the EJTAG PC Trace codes must be aware of the contents of this register in order to trace the code after the vectored interrupt is taken.

For probes that do not support a 4-bit exception code, the LX4380 can be configured via the EJTAG_XV_BITS lconfig option to use only the 3-bit standard codes. In that case, if a vectored interrupt is taken, the 3-bit code for RESET will be presented.

1. Based on EPI Interface Specifications for MAJIC™ and MAJIC PLUS ™

10.2.6. Demultiplexing of TDO and TDI During PC Trace

Normally, EJTAG TDI and TDO are multiplexed with the debug interrupt (DINT) and TPC[0] when in PC Trace mode. This reduces the number of pins required by PC Trace, but prevents any access to EJTAG registers during PC Trace.

To allow access to EJTAG registers during PC Trace, and to facilitate PC Trace in multiprocessor environments, the *lconfig* option JTAG_TRST_IS_TPC=YES causes TDI and TDO to be de-multiplexed such that TRST is used as TPC[0] and DINT is generated via EJTAG registers.

Appendix A. Instruction Formats

This appendix documents the LX4380 instruction encodings that are not included in the standard MIPS-I (R2000/R3000) instruction set.

A.1. Major Opcodes

A.2. Major Opcodes

Table 43: Major Opcode Instruction Formats

	31 26	25 21	20 16	15 6
Assembler Mnemonic	**Major Opcode**	**rS**	**rT**	**Immediate**
CACHE	CACHE	base	op	offset
user defined	CE1IMM	rS	rT	user defined
	6	5	5	16

Table 44: Major Opcode Bit Encodings

Inst[31:29] \ Inst[28:26]	0	1	2	3	4	5	6	7
0	SPECIAL							
1								
2								
3	CE1IMM	CE1IMM	CE1IMM	CE1IMM	SPECIAL2		LEXOP2	
4								
5								CACHE
6								
7					LEXOP			

A.3. LEXOP Instructions

Table 45: LEXOP Instruction Formats

	31 26	25 21	20 16	15 6	5 0
Assembler Mnemonic	**LEXOP 111 100**	**rS**	**rT**	**Immediate**	**LEXOP Subop**
MADH	LEXOP	rS	rT	0000000000	MADH
MADL	LEXOP	rS	rT	0000000000	MADL
MAZH	LEXOP	rS	rT	0000000000	MAZH
MAZL	LEXOP	rS	rT	0000000000	MAZL
MSBH	LEXOP	rS	rT	0000000000	MSBH
MSBL	LEXOP	rS	rT	0000000000	MSBL
MSZH	LEXOP	rS	rT	0000000000	MSZH
MSZL	LEXOP	rS	rT	0000000000	MSZL
	6	5	5	10	6

Table 46: LEXOP Subop Bit Encodings

	Inst[2:0]							
Inst[5:3]	0	1	2	3	4	5	6	7
0	MADH		MADL		MAZH		MAZL	
1								
2								
3								
4	MSBH		MSBL		MSZH		MSZL	
5								
6								
7								

A.4. LEXOP2 Instructions

Table 47: LEXOP2 Load Instruction Formats

	31 26	25 21	20 16	15 6	5 0
Assembler Mnemonic	**LEXOP2 011 110**	**rS**	**rT**	**Immediate**	**LEXOP2 Subop**
LTW	LEXOP2	rS	rT-even, 0	displacement/8	LTW
	6	5	5	10	6

base, rT	Selects general register r0 - r31.
rT-even	Selects general register even-odd pair r0/r1, r2/r3 ... r30/r31.
displacement	Signed 2s-complement number in bytes.

Table 48: LEXOP2 Subop Bit Encodings

Inst[5:3] \ Inst[2:0]	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7					LTW			

A.5. COP0 Instructions

Table 49: COP0 Instruction Formats

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**COP0 010 000**	**rS**	**rT**	**rD**	**0**	**COP0 Subop**
MFLXC0	COP0	MFLX 00011	rS	rD	00000	LXC0
MTLXC0	COP0	MTLX 00111	rS	rD	00000	LXC0
DERET	COP0	00000	00000	00000	00000	DERET
	6	5	5	5		11

These encodings are variants of the standard MTC0 and MFC0 instructions that allow access to the Lexra Coprocessor 0 registers listed below. As with any CP0 instruction, a Coprocessor Unusable Exception is taken in User mode if the Cu0 bit is 0 in the CP0 Status register when these instructions are executed.

rT Selects general register r0 - r31.
rD Selects Lexra Coprocessor 0 register:
00000 ESTATUS
00001 ECAUSE
00010 INTVEC
00011 CVSTAG (for Lexra diagnostic purposes only)
001xx reserved
01xxx reserved
1xxxx reserved

Table 50: COP0 Subop Bit Encodings

Inst[5:3] \ Inst[2:0]	0	1	2	3	4	5	6	7
0	LXC0							
1								
2								
3								DERET
4								
5								
6								
7								

A.6. SPECIAL Instructions

Table 51: SPECIAL Instruction Formats

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**SPECIAL 000 000**	**Copz rs**	**rt**	**rd**	**0**	**SPECIAL Subop**
MOVN	SPECIAL	rS	rT	rD	00000	MOVN
MOVZ	SPECIAL	rS	rT	rD	00000	MOVZ
user defined	SPECIAL	rS	rT	rD	00000	CE1REG
	6	5	5	5	5	6

Table 52: SPECIAL Subop Bit Encodings

	Inst[2:0]							
Inst[5:3]	0	1	2	3	4	5	6	7
0								
1			MOVZ	MOVN				
2								
3								
4								
5								
6								
7	CE1REG		CE1REG	CE1REG	CE1REG		CE1REG	CE1REG

A.7. SPECIAL2 Instructions

Table 53: SPECIAL2 Instruction Formats

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**SPECIAL2 000 000**	**Copz rs**	**rt**	**0**	**0**	**SPECIAL2 Subop**
MAD	SPECIAL2	rS	rT	00000	00000	MAD
MADU	SPECIAL2	rS	rT	00000	00000	MADU
MSUB	SPECIAL2	rS	rT	00000	00000	MSUB
MSUBU	SPECIAL2	rS	rT	00000	00000	MSUBU
SDBBP	SPECIAL2	00000	00000	00000	00000	SDBBP
	6	5	5	5	5	6

Table 54: SPECIAL2 Subop Bit Encodings

	Inst[2:0]							
Inst[5:3]	0	1	2	3	4	5	6	7
0	MAD	MADU			MSUB	MSUBU		
1								
2								
3								
4								
5								
6								
7								SDBBP

Appendix B. Lconfig Forms

B.1. Configuration Options for the LX4380 Processor

This section provides a summary of the configuration options available with *lconfig*. Refer to *lconfig* forms for a detailed description of these form options.

Table 55: Configuration Options

***Lconfig* Option**	**Description**
CBI_WBUF	CBUS Interface write buffer depth
CE0	custom engine 0
CE1	custom engine 1
CLOCK_BUFFERS	clock buffers at top-level module
COP1	coprocessor interface 1
COP2	coprocessor interface 2
DCACHE	data cache size
DCACHE_POLICY	data cache writeback/writethrough policy selection
DMEM	local scratch pad data RAM
EJTAG	EJTAG Debug Support
EJTAG_DATA_BREAK	Number of data breaks to be compiled
EJTAG_DCLK_N	EJTAG PCTrace DCLK N parameter
EJTAG_INST_BREAK	Number of instruction breaks to be compiled
EJTAG_TPC_M	EJTAG PCTrace TPC M parameter
EJTAG_XV_BITS	EJTAG PCTrace number of Exception Vector bits
ICACHE	instruction cache size
IMEM	local instruction RAM with line valid bits
JTAG	Internal JTAG Tap controller with EJTAG support
JTAG_TRST_IS_TPC	TRST pin is TPC out, instead of TDO/TPC mux
LBC_RBUF	Lexra Bus Controller read buffer depth
LBC_RDBYPASS	Lexra Bus Controller read bypass enable
LBC_SYNC_MODE	LBC synchronous/asynchronous selection
LINE_SIZE	cache line size, in words

***Lconfig* Option**	**Description**
LMI_RANGE_SOURCE	source of LMI address ranges
MEM_FIRST_WORD	cache line fill first word
MEM_LINE_ORDER	cache line fill beat ordering
MMU	Memory Management Unit implementation
PC_TRACE	EJTAG PC trace pins
PRODUCT	Lexra Processor name
RESET_BUFFERS	reset buffers at top-level module
RESET_MODE	flip-flop reset method
SCAN_INSERT	Controls scan insertion and synthesis
SCAN_MIX_CLOCKS	scan chains can cross clock boundaries
SCAN_NUM_CHAINS	number of scan chains
SCAN_SCL	scan collar insertion on RAM interfaces
SEN_BUFFERS	scan enable buffering
SEN_DIST	scan enable distribution method
SIM_RAM_TECH	simulate with technology specific RAM wrappers
SIM_TECH	control use of technology specific files in simulation
SYNTH_TECH	control use of technology specific libraries in synthesis
SYSTEM_INTERFACE	system bus interface type
TECHNOLOGY	identifies target technology
TLB_ENTRIES	number of entries in Translation Lookaside Buffer
UNIQUE_NAME	apply a unique name to the RTL
WRITETHROUGH_RANGE	writethrough range for writeback data cache

Appendix C. Port Descriptions

Table 56 shows the possible port connections for the top level module of the LX4380 processor, known as lx2. The actual lx2 ports that are present depends upon *lconfig* settings.

Port names that include a trailing _N or intermediate _N_ indicate active low signals. All other signals are active high unless otherwise indicated.

All input ports must be connected to valid logic-level sources.

The information in the table's Timing column indicates the point within a cycle when the signal is stable, in terms of percent. The Timing column also includes parenthetical references to these notes:

1. Clocked in the JTAG_CLOCK domain.
2. Clocked in the BUSCLK domain if is asynchronous. Otherwise, clocked in the SYSCLK domain.
3. Does not require a constraint (e.g., a clock).
4. A constant that is treated as a false path for timing analysis. These inputs must not change after the processor is taken out of reset.
5. Timing is specified with a symbol in techvars.scr script (e.g. RAM timing).
6. A test-related input or output that is treated as false path for timing analysis. Such inputs must not change during normal at-speed operation.
7. An asynchronous input.

If no clock domain is specified, the signal is clocked in the SYSCLK domain.

For single bit signals, the signal name and signal description indicate the action or function when the signal is in the active state.

Table 56: LX4380 Processor Port Summary

Port Name	I/O	Timing	Description
Clocking, Reset, Interrupts and Control			
SYSCLK	input	(3)	Processor clock.
ResetN	input	10%	Warm reset (or reset "button"), active low.
CResetN	input	10%	Cold reset (or power on), active low.
RESET_D1_R_N	input	30%	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.

Port Name	I/O	Timing	Description
RESET_D1_BR_N	input	30%	BUSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_PWRON_C1_N	input	30%	Power on reset copy for JTAG.
RESET_PWRON_D1_LR_N	input	30%	SYSCLK domain power on reset for EJTAG.
RESET_D1_R_N_O	output	30%	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_D1_BR_N_O	output	30%, (2)	BUSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_PWRON_C1_N_O	output	30%	Power on reset copy for JTAG.
RESET_PWRON_D1_LR_N_O	output	30%	SYSCLK domain power on reset for EJTAG.
INTREQ_N[15:2]	input	(7)	Interrupt requests (level sensitive, active low).
EXT_HALT_P	input	50%	External stall line. Tie to 0 if not used. 1 - stall pipeline next cycle 0 - advance pipeline if no internal stalls
Configuration			
CFG_TLB_DISABLE	input	(4)	Disable TLB mappings even if the TLB is present.
CFG_HLENABLE	input	(4)	Strap to one to enable internal HI/LO registers.
CFG_MACENABLE	input	(4)	Strap to one to enable internal MAC (if present).
CFG_MEMSEQUENTIAL	input	(4)	Strap to one if line reads return words in sequential order, zero if interleave order.
CFG_MEMZEROFIRST	input	(4)	Strap to one if line reads return word zero first, zero if desired word first.
CFG_LBCWBDISABLE	input	(4)	Strap to one to disable read bypass of LBC write buffer, zero to allow read bypass.
CFG_EJTNMINUS1[1:0]	input	(4)	Strap with EJTAG DCLK N minus 1 configuration (0-3=1-4).
CFG_EJTMLOG2[1:0]	input	(4)	Strap with EJTAG M log2 (0-3=1,2,4,8) configuration.
CFG_EJT3BITXVTPC	input	(4)	Strap with ETJAG 3-bit TPC configuration.
CFG_EJTBIT0M16	input	(4)	Strap with EJTAG PC bit0 in TPC configuration.
CFG_DWBASE[31:10]	input	30%	Strapped with DMEM base address configuration value.
CFG_DWTOP[23:10]	input	30%	Strapped with DMEM top address configuration value.
CFG_IWBASE[31:10]	input	30%	Strapped with IMEM base address configuration value.
CFG_IWTOP[`23:10]	input	30%	Strapped with IMEM top address configuration value.

Port Name	I/O	Timing	Description
CFG_DWDISW	input	(4)	Strap to one to disable processor DMEM writes. Must be zero for LX4380.
Test and Debug			
JTAG_RESET_O	output	20%, (1)	JTAG is in TEST-LOGIC-RESET state, active low.
JTAG_RESET	input	(6)	JTAG is in TEST-LOGIC-RESET state, active low.
TAP_RESET_N_O	output	20%, (1)	TAP controller reset.
TAP_RESET_N	input	(6)	TAP controller reset.
JTAG_TDO_NR	output	50%, (1)	Test data out, active low.
JTAG_TDI	input	60%, (1)	Test data in.
JTAG_TMS	input	60%, (1)	Test mode select.
JTAG_CLOCK	input	(3)	Test clock.
JTAG_TRST_N	input	(6)	Test reset.
JTAG_CAPTURE	output	20%, (1)	JTAG is in DATA REGISTER CAPTURE state
JTAG_SCANIN	output	50%, (1)	Scan input to chain
JTAG_SCANOUT	input	50%, (1)	Scan output from chain
JTAG_IR[4:0]	output	20%, (1)	Contents of INSTRUCTION REGISTER
JTAG_SHIFT_IR	output	20%, (1)	JTAG is in SHIFT INSTRUCTION REGISTER state
JTAG_SHIFT_DR	output	20%, (1)	JTAG is in SHIFT DATA REGISTER state
JTAG_RUNTEST	output	20%, (1)	JTAG is in RUN-TEST state
JTAG_UPDATE	output	20%, (1)	JTAG is in DATA REGISTER UPDATE state
EJC_ECRPROBEEN_R	output	30%	One indicates EJTAG probe is active.
JPT_PCST_DR[M-1:0]	output	30%	EJTAG PC trace status; M= 1, 2, 4 or 8.
JPT_TPC_DR(N*3-1:0]	output	30%	EJTAG PC trace value, N= 1, 2, 3 or 4.
JPT_DCLK	output	(3)	EJTAG PC trace clock.
SEN	input	(6)	Scan enable, active high.
TMODE	input	(6)	Test mode, active high.
SIN[<k>:0]	input	(6)	Scan Input. <k> can range from 7 to 0.
SOUT[<k>:0]	output	(6)	Scan Output. <k> can range from 7 to 0.
LBC Interface (to LBus)			
LAddrO[31:0]	output	20%, (2)	Address.
LCmdO[8:0]	output	20%, (2)	Output command.
LDataO[63:0]	output	20%, (2)	Output data.

Port Name	I/O	Timing	Description
LDataI[63:0]	input	50%, (2)	Input data.
LIrdyO	output	20%, (2)	LBC initiator ready.
LIrdyI	input	30%, (2)	System initiator ready.
LFrameO	output	20%, (2)	LBC transaction frame.
LFrameI	input	30%, (2)	System transaction frame.
LSel	input	30%, (2)	System slave select.
LTrdyI	input	30%, (2)	System target ready.
LId	output	20%, (2)	Instruction/data.
LUc	output	20%, (2)	1 - Uncacheable transfer. 0 - Cachable transfer.
LCoe[9:0]	output	20%, (2)	Command output enable. Identical copies are provided to relieve the fanout.
LToe	output	20%, (2)	Transaction output enable.
LDoe[7:0]	output	20%, (2)	Data output enable. Identical copies are provided to relieve the fanout.
LReq	output	50%, (2)	Bus request.
LGnt	input	30%, (2)	Bus grant.
Coprocessor Interface <z=1,2>			
Czcondin	input	80%	Cop branch flag.
Czrd_addr[4:0]	output	50%	Cop read address.
Czrhold	output	45%	Cop hold condition, one stalls coprocessor.
Czrd_gen	output	50%	Cop general register read command.
Czrd_con	output	50%	Cop control register read command.
Czrd_data[31:0]	input	80%	Cop read data.
Czwr_addr[4:0]	output	20%	Cop write address.
Czwr_gen	output	20%	Cop general register write command.
Czwr_con	output	20%	Cop control write address command.
Czwr_data[31:0]	output	30%	Cop write data.
Czinvld_M	output	60%	Cop invalid instruction flag, one indicates invalid instruction in M stage.
Czxcpn_M	output	60%	Cop exception flag, one indicates exception in M stage.

Port Name	I/O	Timing	Description
Custom Engine Interface			
CEI_CE1HOLD	output	45%	CPU is halting Custom Engine.
CEI_CE1INVLD_M	output	40%	Instruction is not valid, M stage.
CEI_CE1INVLDP_S_R	output	30%	Instruction is not valid, S stage.
CEI_XCPN_M_C1	output	40%	CPU reports exception.
CEI_CE1OP_S_R[11:0]	output	30%	Custom Engine op code.
CEI_INSTM32_S_R_C1_N	output	30%	One indicates 32-bit instruction mode; zero indicates 16-bit instruction mode.
CEI_CE1AOP_E_R[31:0]	output	35%	A operand.
CEI_CE1BOP_E_R[31:0]	output	35%	B operand.
CE1_RES_E[31:0]	input	45%	Result from Custom Engine.
CE1_SEL_E_R	input	30%	One indicates Custom Engine opcode is present in E stage.
CE1_HALT_E_R[2:0]	input	20%	Custom Engine stalls processor by driving to ones, allows processor to run by driving to zeros. (Copies must be supplied from multiple registers to meet timing requirements.)
CBUS Interface			
CBUS_YREQO	output	20%	0 - no request present,1 - request present.
CBUS_YADDRO[31:0]	output	20%	Address
CBUS_YREADO	output	20%	1=Read, 0=Write
CBUS_YSZO[3:0]	output	20%	Transfer size 4'b1000 - byte 4'b1001 - 2 bytes 4'b1011 - word 4'b1101 - 2 words 4'b0000 - 4 words
CBUS_YLINEO	output	20%	1=line access, 0=single access.
CBUS_YDATAO[63:0]	output	20%	Write Data
CBUS_YUCO	output	20%	1=uncached, 0=cached access.
CBUS_YSRCO[3:0]	output	20%	transaction source (within LX4380): 4'b0001 Instruction Cache 4'b0010 Data Cache or EJTAG DMA write 4'b0100 EJTAG DMA read 4'b1000 not used
CBUS_YDBUSYO	output	20%	1 - LX4380 is not ready to receive data for a Data Read. Any return read data with VAL-TYPE of Data Read will be ignored by the LX4380. External logic must hold such data CBUS_YDBUSYO is de-asserted. 0 - LX4380 is ready to receive data.

Port Name	I/O	Timing	Description
CBUS_YBUSYI	input	80%	1 - External logic cannot accept request. External logic ignores any current request. 0 - External logic is ready to accept a request.
CBUS_YDATAI[63:0]	input	80%	Read Data.
CBUS_YVALTYPEI[3:0]	input	80%	Indicates valid read data of a certain type: 4'b0000 No valid read data 4'b0001 Instruction Cache 4'b0010 Data Cache 4'b0100 EJTAG DMA 4'b1000 not used
CBUS_YIDLEI	input	80%	Indicates external CBUS_Y device has no pending read or write transactions.

Appendix D. Pipeline Stalls

This appendix documents the stall conditions that may arise in the LX4380.

D.1. Stall Definitions

Issue stall: an invalid instruction enters each pipe, while any other valid instructions in the pipe advance.

Pipeline stall: All instructions in the pipe stay in the same stage, and do not advance.

Stall: if not otherwise qualified, means Pipeline stall.

D.2. Instruction Groupings

Table 57: Instruction Groupings For Stall Definition

Group Name	Instructions In Group
M-I-LoadStore	LB, LH, LW, LBU, LHU, LWC1, LWC2, LWC3 SB, SH, SW, SWC1, SWC2, SWC3
M-I-Mac	MULT(U), DIV(U), MFHI, MFLO, MTHI, MTLO MADH, MADL, MAZH, MAZL MSBH, MSBL, MSZH, MSZL
M-I-Control	J, JAL(X), JR, JALR BLTZAL, BGEZAL (linked branches) SYSCALL, BREAK All COPz (MFCz, CFCz, MTCz, CTCz, BCFz, BCTz, RFE) LWCz, SWCz (also in LoadStore group) MTLXC0, MFLXC0 (Lexra-specific)
M-I-UnlinkedBranch	BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ
M-I-General	All remaining M-I instructions
MIV-CMove	MOVZ, MOVN
NVX-LoadStore	LTW
EJTAG-Control	DERET, SDBBP

D.3. Non-Sequential Program Flow Issue Stalls

M-I JR,JALR

Two issue stalls after the delay slot instruction.

M-I J, JAL(X), and M-I taken branches:

NO stall cycles after the delay slot instruction.

M-I not-taken branches

Two issue stalls after the delay slot instruction.

The branch rules are a consequence of the fact that all branches are predicted to be taken.

D.4. Load/Store Rules

Load-Use A-Stage Single Cycle Pipeline Stall:

After a Load instruction to a target register, an instruction which follows the load in the pipeline by two cycles and uses that target register of the load will pipeline stall for one cycle.

For Twinword Loads (LTW) this rule applies to both of the target registers in the register-pair operand.

Store-Load Data RAM Access Stall:

A Load instruction which follows a Store instruction by two cycles always causes a one-cycle stall.

Note: This stall only applies if the Store instruction hits in the data cache.

Store-Store Tag RAM Access Stall:

A second Store instruction which follows a first Store instruction by two CYCLEs causes a one-cycle stall IF the first Store is to a previously Clean line of a Write-Back cache.

Note: This stall only applies if the first Store instruction hits in the data cache.

Store-Load Data Read-After-Write Stall:

A Load instruction which follows a Store instruction by one CYCLE causes a two-cycle stall IF the Load accesses data at the same word address as the Store.

For Twinword load instructions, either of the load word addresses may match the Store word address.

Store-Store Tag-DirtyBit Read-After-Write:

No stall.

Hardware detects the case of back-to-back stores to the same line and eliminates any replay of the second store to access the Tag-DirtyBit.

Store-Load Tag Invalidate Tag RAM Access Stall:

A Store or Load instruction that follows by two CYCLEs an uncached Store or Load instruction that causes a TAG invalidate causes a one-cycle stall.

Store-Load Tag Invalidate Read-After-Write Stall:

A Store or Load instruction that follows by one CYCLE an uncached Store or Load instruction that causes a TAG invalidate causes a two-cycle stall, IF the second instruction accesses data in the same

cache line as the first instruction.

D.5. Mac Ops interlock matrix

The Mac eliminates all programming hazards between Mac instructions by stalling the pipeline as necessary. This is done both to avoid resource conflicts as well as to wait for results of a first instruction that is needed by a second instruction.

The following table indicates the number of cycles that must be inserted between the first indicated instruction and the second. A zero (or dash) indicates that the instructions can issue back-to-back to the Mac pipe with no stalls. A non-zero number indicates the number of stall cycles that will occur if the instructions are issued in consecutive cycles. These stall cycles are available for any other non-Mac instructions, but should NOT be filled with NOPs since that would only increase the code footprint without improving performance.

Table 58: Cycles Required Between MAC Instructions

1st Op / 2nd Op	MULT, MULTU, MAD, MADU, MSUB, MSUBU	MTHI, MTLO, MADH, MADL, MAZH, MAZL, MSBH, MSBL, MSZH, MSZL, MFHI, MFLO	DIV, DIVU
MULT, MULTU, MAD, MADU, MSUB, MSUBU, MADH, MADL, MAZH, MAZL, MSBH, MSBL, MSZH, MSZL	1 cycle	-	19 cycles
DIV, DIVU	1 cycle	-	19 cycles
MFHI, MFLO	3 cycles	2 cycles	19 cycles

D.6. MVCz Stall

The coprocessor move instructions (M-I: MTCz, CTCz, LWCz, MFCz, CFCz) are always followed by two cycle issue stalls.

The variants of coprocessor move instructions (MTLXC0, MFLXC0) are always followed by two cycle issue stalls.

The instructions TLBP and TLBR, which update Coprocessor 0 registers, are always followed by two cycle issue stalls.

D.7. TLBW Stall

The TLB write instructions (TLBWI, TLBWR) are always followed by a one cycle issue stall.

D.8. MMU Stalls

ITLB Stall:

When the program jumps, branches, or increments from the most recently used page to another page in the ITLB, a single cycle stall is incurred.

When the program jumps, branches or increments to a page not in the ITLB, a four-cycle stall is incurred if the target VPN is mapped, one-cycle if the target VPN is unmapped.

If the target VPN is not in the joint TLB, an exception is recognized when the instruction reaches the M-stage.

A TLBWI/TLBWR instruction invalidates any ITLB entry corresponding to the over-written joint TLB entry.

ITLB Issue Stall:

When an ITLB stall occurs due to incrementing across a page boundary, AND there is any of the following instructions found anywhere in the last doubleword of the page, then there is one issue stall in addition to the ITLB stalls:

M-I branch of any kind
M-I J, JAL(X)
EJTAG DERET

DTLB Stall:

When a Load or Store uses a base register that is in the DTLB and hits a VPN that is in the DTLB, there is no stall incurred.

When a Load or Store uses a base register that is in the DTLB but does not hit a VPN that is in the DTLB, a two-cycle stall is incurred if the VPN is mapped, one-cycle if the VPN is unmapped.

When a Load or Store uses a base register that is not in the DTLB, a three-cycle stall is incurred if the VPN is mapped, two-cycles if the VPN is unmapped.

Notes on DTLB entry maintenance:

1) A TLBWI/TLBWR instruction invalidates any DTLB entry corresponding to the over-written joint TLB entry.

2) Any instruction that updates a base register invalidates (on the S->E transition) DTLB entries using that register.

3) A DTLB entry that is invalidated per item (2) is resurrected (on the E->A transition) with the new base register value if the invalidating instruction is one of the following:

ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI	(OP[31:29] == 001)
SLL, SRL ,SRA, SLLV, SRLV, SRAV	(SPECIAL+OP[5:3] == 000)
ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR	(SPECIAL+OP[5:3] == 100)

4) When a new DTLB entry is created for a VPN, the replacement policy is FIFO. Bubbles in the

FIFO that occurred because of item (2) are collapsed.

D.9. Cache Miss Stalls

Instruction Cache Miss Stall:

When an instruction cache miss occurs, the processor is stalled for the duration of the cache line fill operation.

The number of cycles required to complete the line fill is system dependent.

Instruction Cache 2-Way Soft Miss Stall:

When a 2-way Icache is in use, a soft-miss is defined as a hit in the unpredicted way, with way prediction defined as follows:

When not running in Lock mode, use the LRU bit.

When running in LockedDown mode, if the most recent LockedDown Icache access hit a Locked line, then predict way 1 (the Locked way), else use the LRU bit.

When running in LockGather mode, predict way 1 (the Locked way). This prevents a “hit” (without soft-miss) on way 0, thus allowing for the invalidation of way 0 (and a fill to way 1) in that case. Also, a “miss” is forced in LockGather mode whenever the Lock state is clear, to allow the Lock state to be set for a way 1 hit (that was not previously locked). A “miss” is never allowed to be “soft” in LockGather mode, which forces the fill to way 1 in the case of a way 0 hit as noted above.

A soft miss always causes a two-cycle stall.

Data cache miss stall:

When a data cache miss occurs as the result of a load instruction, the processor stalls while it waits for the data. The data cache releases the stall condition after the required word is supplied to the processor, even if additional words must still be filled into the data cache. However, if the processor issues another load or store operation to the data cache while the remainder of the line fill is in progress, the cache will again stall the processor until the line fill operation is completed.

The number of cycles required to complete the line fill is system dependent.

Evict Buffer Not-Empty Stall:

When a data access (load or store) needs to use the system bus and the Evict Buffer is not empty due to a previous evict operation, the processor stalls while it waits for the evict buffer to empty.

D.10. Pipeline Diagrams for Non-Sequential Program Flow Issue Stalls

M-I JR, JALR:

```
JR          I  D  S  E  A  M  W
delayslot      I  D  S  E  A  M  W
notvld            I  .  .  .
notvld               I  .  .
target                  I  D  S  E  A
```

M-I J, JAL(X), and M-I Taken Branches:

```
J          I  D  S  E  A  M  W
delayslot     I  D  S  E  A  M  W
target           I  D  S  E  A  M
```

M-I Not-Taken Branches:

```
B-ntkn     I  D  S  E  A  M  W
delayslot     I  D  S  E  A  M  W
notvld           I  .  .  .
notvld              I  .  .  .
delay+4                I  D  S
```

Load-Use A-stage Single Cycle Pipeline Stall:

```
00: lw s0,0(a0)   I  D  S  E  A  M
04: addi a0,4        I  D  S  E  A  A  M  W
08: add s1,s0           I  D  S  E  E  A  M  W
0c: add t1,t2              I  D  S  S  E  A  M  W

    RHOLD                     X
    DLOAD_M                   X
```

Store-Load Data RAM Access Stall:

```
00: sw s0,00(a0)   I  D  S  E  A  M  W
04: foo               I  D  S  E  A  M  M  W
08: lw s2,32(a0)         I  D  S  E  A  A  M  W

    RHOLD                        X
```

Store-Store Tag RAM Access Stall:

```
00: sw s0,00(a0)   I  D  S  E  A  M  W
04: foo               I  D  S  E  A  M  M  W
08: sw s2,32(a0)         I  D  S  E  A  A  M  W

    RHOLD                        X
```

Store-Load Data Read-After-Write Stall:

```
00: sw s0,00(a0)   I  D  S  E  A  M  W
04: lw s2,00(a0)      I  D  S  E  A  A  A  M  W

    RHOLD                     X  X
```

D.11. Pipeline Diagram for Mac Ops Interlock Stall

```
00: mult s0,s1    I  D  S  E  A  M  -
04: lw s0,0(a0)      I  D  S  E  A  M  M  M  W
08: lw s1,0(a0)         I  D  S  E  A  A  A  M  M  M  W
0c: mflo v0                I  D  S  E  E  E  A  M  W
10: sw v0,0(a1)               I  D  S  S  S  E  A  M  W

    multcount(4S)             0  1  2  3  4
    RHOLD                        X  X
```

D.12. Pipeline Diagram for MVCz Stall

```
00: mtc0       I  D  S  E  A  M  W
04: foo           I  d  d  D  S  E  A  M  W
08: foo1                  I  D  S  E  A  M  W
```

D.13. Pipeline Diagram for TLBW Stall

The handler for a TLB exception can return to the offending instruction after writing a new JTLB entry with the following canonical code fragment:

```
00: tlbwr      I  D  S  E  A  M  W
04: jr            I  d  D  S  E  A  M  W
08: rfe              I  D  S  E  A  M  W
0c: foo                 I  D  .  .
10: foo                    I  .  .
tgt:                          I  I  I  I  I  D
          ITLB-REQUEST        X
          JTLB-RESPONSE             X
          SELECT NEW PFN TO RAM        X
```

The target of the JR can use (for its Ifetch) the newly created JTLB entry that is written in the W-stage. This is due to the single issue stall after the TLBW, and the fact that the JR target address is resolved in the E-stage of the JR. It is also true that any Data access in the target or subsequent instructions can use the newly created JTLB entry.

D.14. Pipeline Diagrams for DTLB Stalls

Base assumption, all cases: DTLB entry exists for LW r1, 0(r2) where r2 is page aligned.

CASE 1: no stall

```
00: lw r1,4(r2)    I  D  S  E  A  M  W

DTLB_HIT_S            X
```

CASE 2: reg-hit, VPN-miss, VPN mapped, create new entry

```
00: lw r1,-4(r2)   I  D  S  E  E  E  A  M  W
04: lw r3,-8(r2)      I  D  S  S  S  E  A  M  W

DTLB_REGHIT_S            X        X
DTLB_VPNHIT_S            -        X
```

CASE 3: reg-miss, VPN mapped, create new entry

```
00: lw r1,-4(r2)   I  D  S  E  E  E  E  A  M  W
04: lw r3,-8(r2)      I  D  S  S  S  S  E  A  M  W

DTLB_REGHIT_S            -           X
DTLB_VPNHIT_S            .           X
```

CASE 4: reg-invalidate, VPN mapped

```
00: lw r2,0(r2)    I  D  S  E  A  M  W
04: foo               I  D  S  E  A  M  W
04: lw r3,0(r2)          I  D  S  E  E  E  E  A  M  W

DTLB_REGHIT_S                  -
DTLB_VPNHIT_S                  .
```

CASE 5: reg-invalidate and resurrect, no stall

```
00: addiu r2,r2,4  I  D  S  E  A  M  W
04: foo               I  D  S  E  A  M  W
04: lw r3,0(r2)          I  D  S  E  A  M  W

DTLB_REGHIT_S                  X
DTLB_VPNHIT_S                  X
```

CASE 6: Vector Add C=A+B, no stalls

After initialization, DTLB entries valid for C(base r1), A(base r2), B(base r3) all initially page aligned.

```
00: sw r7,0(r1)    I  D  S  E  A  M  W
04: addiu r1,r1,4     I  D  S  E  A  M  W
08: lw r5,0(r2)          I  D  S  E  A  M  W
0c: addiu r2,r2,4           I  D  S  E  A  M  W
10: lw r6,0(r3)                I  D  S  E  A  M  W
14: addiu r3,r3,4                 I  D  S  E  A  M  W
18: bne r3,r9,00:                    I  D  S  E  A  M  W
1c: add r7,r5,r6                  I  D  S  E  A  M  W

DTLB_REGHIT_S            X     X     X
DTLB_VPNHIT_S            X     X     X
```

D.15. Pipeline Diagrams for Cache Misses

Instruction Cache Miss Stall:

```
08: foo0          I  D  S  E  A  A  A  A  A  A  M  W
0c: foo1             I  D  S  E  E  E  E  E  E  A  M  W
10: foo2                I  ~d  .  .  .  I  D  S  E  A  M  W

    RHOLD                     X  X  X  X  X
```

Instruction Cache 2-Way Soft Miss Stall:

```
08: foo0          I  D  S  E  A  A  A  M  W
0c: foo1             I  D  S  E  E  E  A  M  W
10: foo2                I  ~d  I  D  S  E  A  M  W
14: foo3                      I  D  S  E  A  M  W

    RHOLD                     X  X
```

Data Cache Miss Stall:

```
04: lw            I  D  S  E  A  M  .  .  .  .  W
08: foo1             I  D  S  E  A  M  M  M  M  M  W
0c: foo2                I  D  S  E  A  A  A  A  A  M  W

    RHOLD                           X  X  X  X
```

Index

Note: All instructions are listed under the “instructions” heading.