LX8380 Data Sheet

Lexra, Inc.

Revision 1.4

November 21, 2001

Lexra Proprietary and Confidential

LX8380 Data Sheet Revision 1.4.

Table of Contents

List of Tables

List of Figures

1. Product Overview

1.1. Introduction

This data sheet describes Lexra's LX8380 processor core, a RISC network processor developed for Intellectual Property (IP) licensing. The LX8380 is a carefully engineered extension to the industry-standard MIPS-I® ISA. The major subsystems are: the CPU core, Local Memory Interfaces (LMI), the Block Move Controller (BMC) and LBus Controller (LBC). The technology includes optional interfaces to a customer-defined Coprocessor (CI2) and Custom Engine (CE) that provide extensions to the MIPS ISA. The local instruction memories and data memories may include caches and fixed RAM; the sizes are configurable. The figure also highlights the LX8380 multi-context register file to support fast context switching. Additional LX8380 extensions include new bit-field operations for efficient packet header processing.

Network communications systems are characterized by demanding, real-time performance requirements. Typically, system designers have addressed these requirements with custom ASICs, off-the-shelf processors, and PLDs. The explosive growth in the size and bandwidth of the Internet has recently stimulated semiconductor companies to develop a new type of product, called a Network Processor Unit (NPU), to serve these applications. These ICs incorporate multiple programmable cores and specialized peripherals. Compared to ASIC development, NPUs offer the system designer faster time-to-market and flexibility to implement differentiated services in software; compared to general-purpose, off-the-shelf components, NPUs offer the promise of lower cost and superior performance through architectural specialization. LX8380 is a scalable processor with the specialized architectural features needed for high-performance packet processing for a wide variety of new products.

The time required to process packets for IP routing and classification is dominated by long latency operations, such as table lookups from large memories and buffer accesses. However, a distinguishing feature of network communications systems is that subsequent packets are readily available for independent processing. Therefore, a fast context switch can be exploited to hide the memory latency. LX8380 includes a configurable number (1-8) of general register sets and program counters, along with instructions for fast context switching. This enables multiple software threads to efficiently execute on a single processor. A thread is de-activated under software control either (i) unconditionally, (ii) when a load with context switch instruction is coded for a long latency load, or (iii) when a command is written to a shared system device.

Following a context switch, the CPU activates a new thread from the pool of ready threads. The context switch does not introduce stall cycles. Because the new thread has an independent general register set, it can quickly resume processing. To avoid stalling the new thread while the previous thread's data transfer completes, the LX8380 incorporates a dedicated port to the processor's data memory for the transfer of packet data. In addition, the memory system is non-blocking, permitting local accesses and cache hits to operate in parallel with one outstanding global access per context. With this architecture, context switches may be used frequently to achieve optimal performance.

Packet processing also requires frequent access to bit-fields in the packet header that are not byte-aligned. For this reason, LX8380 has extended the MIPS-I® Instruction Set Architecture (ISA) to include a complete set of bit-field operations for field extract, insert, set and clear. Deterministic allocation of real-time is another important problem in network communications software. This problem is compounded by multi-processing. For this reason, the LX8380's configuration options include dedicated (uncached) local instruction and data memories for real-time critical instructions and data in order to avoid cache miss penalties.

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 LX8380. Configuration options include Enhanced JTAG (EJTAG) support for debug and In-Circuit Emulation (ICE).

Because the LX8380 executes the MIPS instruction set, 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 (except unaligned loads, stores).
  - Executes Lexra's network processing extensions.
    - High performance context switch.
    - Bit field manipulation.
    - Dual one's complement addition.
    - Hash key formation.
    - Jump tables.
  - Extensive third-party tool support.

- High-Performance Context Switch
  - Processor provides 1-8 contexts (the number is customer-configurable).
  - Independent program counter, status, and general registers for each context.
  - No wasted cycles for context switch.
  - Context switch initiated by program.
  - Thread re-activation based on completion of data transfer, asynchronous external events or program control.

- **System Level Building Blocks**
  - Simplified MMMU (SMMU)
  - 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 Block Move Controller (BMC)
  - Optional EJTAG Draft 2.0.0 support for debugging.

- **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. LX8380 Processor Overview

The LX8380 is a RISC processor that executes the MIPS-I instruction set[1] along with Lexra's networking 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 LX8380 processor.

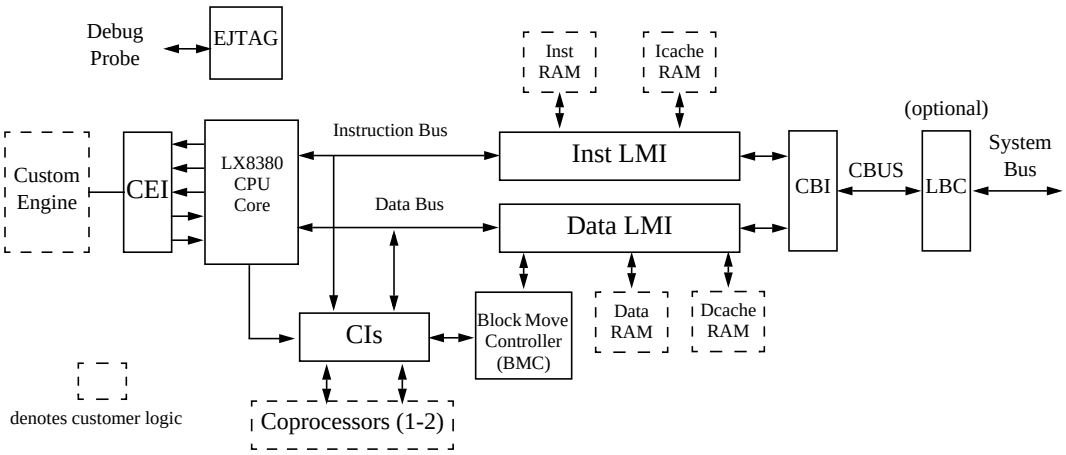


Figure 1: LX8380 Processor Overview

MIPS I Execution. The LX8380 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. An extension to the MIPS ISA allows a pair of 32-bit registers to be loaded from memory. 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.

Context Switching. The LX8380 incorporates up to eight independent 32 x 32b general register sets called contexts. Execution can switch between independent tasks, called threads. This context switch is performed with no wasted cycles and prevents stalls while waiting for data from on-chip or off-chip shared resources. Context switches occur under program control when data is loaded from shared resources. A background load of 32-bits, 64-bits or 128-bits from a shared resource can be accomplished with a single Load

1. The MIPS unaligned load and store instructions (LWL, LWR, SWL, SWR) are not supported.

instruction.

A special class of instructions, called Write Descriptor (WD), allow a command or data to be directed to a shared resource, including a request for up to 128 bits of return data. This allows shared devices to efficiently perform operations that atomically examine and modify memory state. The processor performs the WD operation in a single instruction cycle without stalls by using a context switch. When a context switch occurs, the program counter of the suspended thread is stored in a CP0 register while execution switches to another thread. The next thread is automatically selected from the pool of ready-to-run threads of equal priority, using a windowed round-robin algorithm.

ISA Extensions for Network Processing. Lexra has added 32 new instructions to the LX8380 to optimize for high performance packet processing. Bit-field operations are included to accelerate lookup-key formation used in packet classification. Specialized hash functions, table lookup instructions and one's-complement addition are also included.

Many of the new instructions are used to facilitate high-speed data movement, fundamental to network communications. 64-bits can be loaded from local data RAM into a general register pair in a single cycle. Up to 128-bits can be transferred from shared memory by a single instruction. The Lexra extensions also support atomic read-modify-write operations on the shared memories. Latencies in access to shared memory, on-chip or off-chip, can be hidden using a zero-overhead switch between the eight independent hardware contexts.

Pipeline. LX8380 instructions are executed by a seven-stage pipeline that has been designed so that all transactions internal to the LX8380, 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 LX8380 supports up to eight prioritized, vectored interrupts to meet hard real-time response requirements.

Coprocessor Instructions. The LX8380 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 LX8380 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 LX8380 processor is designed to easily fit into different target applications. It provides the following building blocks.

- A Simplified Memory Management Unit (SMMU) for deeply embedded applications.
- 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.
- An optional Block Move Controller (BMC) supporting up to eight DMA channels for background transfers between local data memory and the system bus.

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

1.3.1. Simple Memory Management Unit (SMMU)

The LX8380 SMMU is designed for embedded applications using a single address space. Its primary function 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.

The optional LX8380 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.

1.3.2. Local Memory Interface (LMI)

The LX8380'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.

1.3.3. 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 LX8380 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 LX8380 sequencer for testing in branch instructions.

1.3.4. Custom Engine Interface (CEI)

The LX8380 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 LX8380. To support multi-cycle operations, a stall input is included in the interface.

1.3.5. Cache Bus (CBUS) Interface

The CBUS interface is a simple signalling layer between the LX8380 processor's cache controllers and the optional LX8380 system bus interface, the LBC. LX8380 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 LX8380 using the CBUS interface.

1.3.6. Lexra Bus Controller (LBC)

The optional Lexra Bus Controller (LBC) is the interface between the LX8380 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 LX8380 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.

1.3.7. Block Move Controller (BMC)

The LX8380’s BMC performs data transfers between the processor’s local data memory and the system bus. These transfers may occur in either direction, and are set up using the coprocessor registers of the BMC. Transfer length may be 1-262,144 (256K) bytes, with byte granularity in the addresses and transfer size. Data transfer takes place in the background without stalling the processor.

The BMC supports up to eight DMA channels and is implemented as an optional Lexra-supplied Coprocessor 3. Completion of block moves can be synchronized with the program using conditional branch instructions, context switches or interrupts.

1.3.8. EJTAG Debug Support

The LX8380 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 LX8380 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 LX8380 core.

1.4. RTL Core & SmoothCore Licensing Models

Lexra delivers LX8380 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 LX8380.

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 LX8380 processor includes the Control Processor (CP0) and the register file and ALU (RALU). CP0 includes instruction address sequencing and exception processing. The RALU performs ALU operations and generates data addresses.

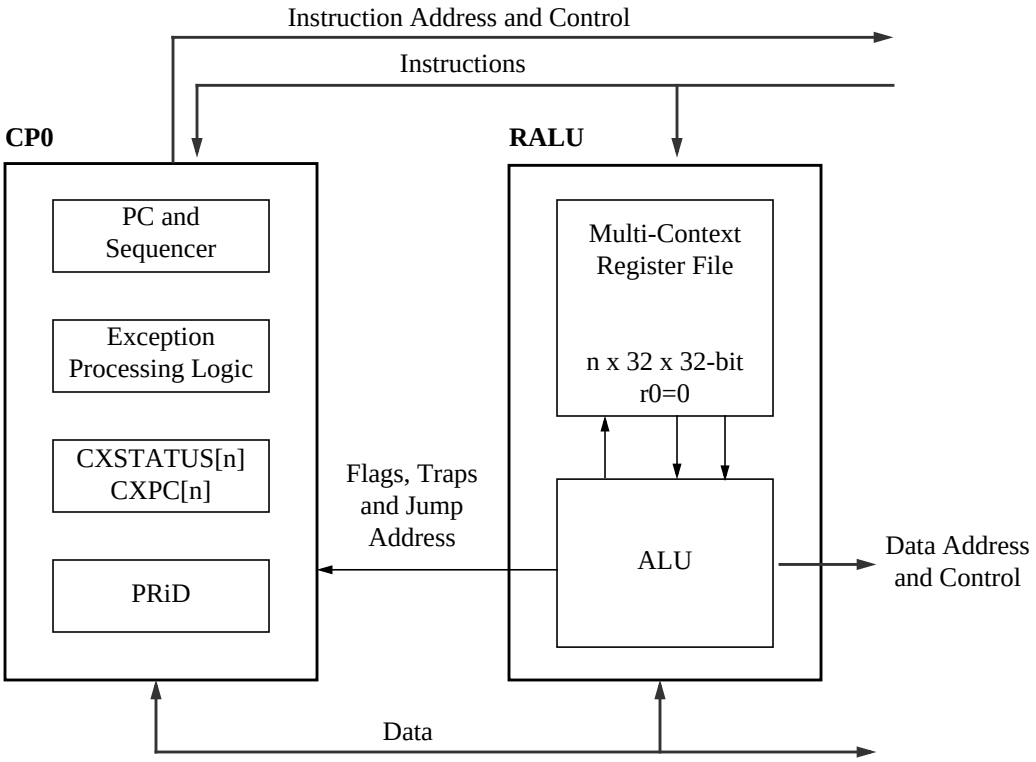


Figure 2: Processor Core Module Partitioning

2.2. Seven Stage Pipeline

The LX8380 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 LX8380'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 LX8380 RALU incorporates a multi-context 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. In the LX8380, the two write ports are also used to support 64-bit loads from the Data Bus.

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 LX8380 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/from the LX8380's Block Move Controller or to 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 LX8380 throughput by several percent. However, the net effect of the LX8380’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
BADVADDR	8	Holds bad virtual address if address exception error occurs
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) 0x0000ce01 for LX8380
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 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 6.2.

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 LX8380 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.

2.5. High-Performance Context Switch

The LX8380 CPU incorporates multiple, independent register sets called *contexts*. As a result, execution can switch between independent software tasks, called *threads*, each running in its own context. This switch is called a *context switch*. Conventional RISC architectures perform context switching in software. However, packet processing demands special hardware support to achieve high performance context switching. The LX8380 provides a zero-overhead context switch. That is, an instruction can be executed for *some* context in every cycle.

2.5.1. New Context Registers

The number of contexts is customer-defined using Lexra's *lconfig* utility. One to eight contexts are supported by the LX8380 RTL (default is one context). Each context includes:

- (32) general registers (r0 - r31)
- (1) 32-bit CXPC (program counter)
- (1) 16-bit CXSTATUS register

The general registers are located in the RALU. The CXPC and CXSTATUS registers are located in CP0. In addition, a 3-bit register MOVECX is located in CP0, and is accessible with the MTLXC0/MFLXC0 instructions (variants of the MIPS standard MTC0/MFC0 instructions). MOVECX holds the encoded number of the target context for the MFCXC/MTCXC and MFCXG/MTCXG instructions, which can access the registers of any context. These new registers are illustrated in Figure 3. The currently active context number is an implicit read-only value that is accessed with the MYCX instruction.

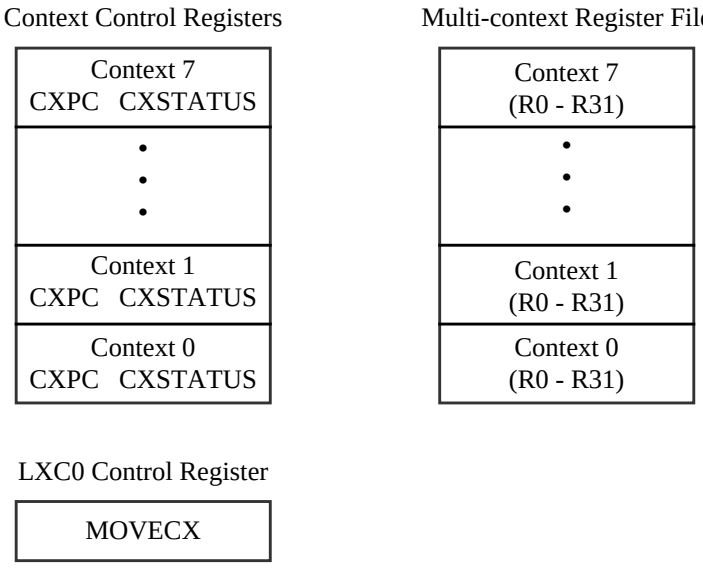


Figure 3: Context Associated Registers

The MIPS I ISA (except for unaligned Loads and Stores) is fully supported in each context. As a result, the general register set for each context is fully consistent with the MIPS ISA requirements. For example, r0 is hard wired to 0 and r31 is an implied "link" for certain branch and jump instructions in every context. Up to two source registers and one destination register may be specified for an ALU operation, again consistent

with the MIPS programming model.

CXPC holds the 32-bit virtual address of the next instruction to be fetched by the associated context. The 16-bit CXSTATUS register indicates whether the context is waiting for data transfer or I/O events. CXSTATUS also permits program-assigned priority for context re-activation.

The CXSTATUS register fields are identified in Table 4. Each field is explained below. The “Rd/Wr” or “Rd Only” indications apply to access using the MTCXC and MFCXC instructions. The effects of other hardware and software events on the fields is shown explicitly and explained in the following paragraphs.

The CXSTATUS WAIT-EVENT field provides eight event flags that may be controlled by hardware, software or a combination of the two. The flags may be set with the CSW instruction or the WD.CSW instruction. The WD.CSW instruction updates the WAIT-EVENT flags, writes a descriptor to the system bus, and performs a context switch.

When WAIT-EVENT bits are set with a WD.CSW instruction, the processor initiates an uncachable write to the system bus, and performs a context switch. All context switches are performed after a one-instruction delay slot. The WAIT-EVENT bits may be cleared via software from another context with the POSTCX instruction, or by hardware through the event signal inputs.

When the target device completes the WD operation, it notifies the processor with a high pulse on the processor’s corresponding event signal input (eight per context). The processor then clears the WAIT-EVENT bit in the context’s CXSTATUS register. Software can set more than one WAIT-EVENT bit, which will require a completion response on each of the corresponding event signal inputs before the context is ready for execution.

The CXSTATUS WAIT-LOAD bit indicates that the associated context is waiting for the completion of a register load from uncached memory (or a memory-mapped I/O) following execution of LW.CSW (load word with context switch), LT.CSW (load twinword with context switch) or LQ.CSW (load quadword with context switch). See Section 2.5.4 for descriptions of these three instructions. WAIT-LOAD is set following execution of LW.CSW, LT.CSW, LQ.CSW, WDLW.CSW, WDLT.CSW or WDLQ.CSW instructions, and cleared by the processor when the load data is transferred to the context’s general register file.

The three-bit THREAD-PRIORITY field in CXSTATUS allows context scheduling with up to eight priorities. An application specific context scheduler can utilize thread priorities to fine tune the context scheduling. See Section 2.5.4 for details of the context scheduling hardware interface.

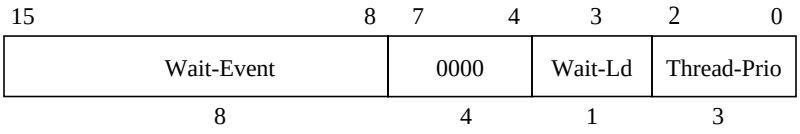


Table 4: Context Status Register Detail

Field	Width (Bits)	Description
WAIT-EVENT	8	(Rd/Wr) Set with CSW and WD.CSW instructions. Cleared by external hardware, or cleared with POSTCX instruction).
Reserved	4	(Rd Only) Reserved.
WAIT-LOAD	1	(Rd/Wr) Set with LW.CSW, LT.CSW, LQ.CSW, WDLW.CSW, WDLT.CSW and WDLQ.CSW instructions. Cleared by hard-ware.
THREAD-PRIORITY	3	(Rd/Wr) Thread (context) priority, for use by optional custom context scheduler.

2.5.2. Reset

At reset,

```
CXSTATUS[15:0]   <—   0x0000
CXPC[31:0]       <—   0xbfc00000
MOVECX[2:0]      <—   000
```

The general registers are unaffected by reset.

Context 0 is activated at reset. All CXPC's are reset to the common MIPS reset vector 0xbfc0000, However, context 0 may modify the initial CXPC of the other contexts prior to the first context switch.

2.5.3. Determining the Number of Contexts in Software

As described above, the number of contexts that are implemented in a processor is customer defined using Lexra's *lconfig* utility. In some cases software will be written that must be adaptable to an unknown number of contexts. For any non-implemented context, reading the CXSTATUS register will always return a value of zero. Using the instructions described in Section 2.5.12, Program Access to New Registers, the software can attempt to write a non-zero value to the CXSTATUS register for each context. If the value zero is returned when attempting to read back the written value, then that context is not implemented.

2.5.4. Initiation of Context Switch

A context switch is executed by the CSW instruction and any of the following instructions that include the .CSW extension:

```
CSW       rs                    context switch, update CXSTATUS from rs
LW.CSW    rt, displacement(base) load word from uncached memory
LT.CSW    rt, displacement(base) load twinword from uncached memory
LQ.CSW    rt, displacement(base) load quadword from uncached memory
WD        rs, rt, device        write descriptor to device
WD.CSW    rs, rt, device        write descriptor to device, with context switch
WDLW.CSW  rd, rs, rt, device    write descriptor, load word reply data
WDLT.CSW  rd, rs, rt, device    write descriptor, load twin reply data
WDLQ.CSW  rd, rs, rt, device    write descriptor, load quad reply data
```

2.5.5. CSW Instruction

The Context Switch (CSW) instruction causes an unconditional context switch, allowing the application program to execute a context switch under complex, program-defined conditions by alternately executing or branching around the CSW instruction. Bits 31:24 of the rs register specified in the CSW instruction are logically OR-ed with the WAIT-EVENT field of CXSTATUS to determine the new WAIT-EVENT field settings.

2.5.6. LW.CSW, LT.CSW and LQ.CSW Instructions

The Load Word with Context Switch (LW.CSW) instruction is used to initiate a long latency transfer from an LBus device to a general register. LW.CSW performs a “split transaction” read so that the next context can continue to execute while the memory-mapped resource is accessed. Only two clock cycles of system bus tenure are required to initiate the split read transaction. Following initiation, the bus is available for other use. The final transfer of the return data uses one cycle of system bus tenure. Loading the final result into the register file will not stall the currently executing context unless the context is executing a load or store instruction at the time the split read data is returned. In this case, a single cycle stall is required to load the split read data into the register file. The currently executing context is otherwise unaffected by the return data.

Similarly, LT.CSW is used to initiate a long latency load of 64-bit data into two consecutively numbered general registers, starting with the low register address bit equal to 0. Up to two processor stalls can occur when the 64-bit data is transferred into the register file. LQ.CSW is used to initiate a long latency load of 128-bit data into four consecutively numbered general registers, starting with the two low order register address bits equal to 00. Up to four processor stalls can occur when the 128-bit data is transferred into the register file.

Following LW.CSW, LT.CSW or LQ.CSW, WAIT-LOAD in CXSTATUS is set.

2.5.7. WD[.CSW] Instructions

The Write Descriptor (WD) instruction forms a 64-bit descriptor from the contents of two general registers, and writes the descriptor over the system bus interface to the specified device. An optional context switch may be performed by this instruction, by appending a .CSW suffix to the mnemonic. These instructions are used to initiate long-latency operations to a shared device.

These instructions form the descriptor using rs and rt register contents, as described in detail in Section 4. For WD.CSW, the upper bits of the descriptor identify the WAIT-EVENT bits to be set. The WD instruction sources the full 64 bits of the descriptor on the system bus. The 32-bit system bus address of the target device is formed by concatenating a 24-bit configuration defined constant, the 5-bit device ID from the instruction opcode and three bits of 0.

2.5.8. WDLW.CSW, WDLT.CSW and WDLQ.CSW Instructions

The WDLW.CSW, WDLT.CSW and WDLQ.CSW instructions provide efficient operation with devices that return 32, 64 or 128 bits of data. These instructions set the WAIT-LOAD bit in the CXSTATUS register. The WDLW.CSW writes a 64-bit descriptor to a device, and requests the device to provide a split transaction word read response. Likewise, the WDLT.CSW (WDLQ.CSW) instruction writes a descriptor and requests the device to provide a split transaction twinword (quadword) read response. Note that a .CSW suffix is mandatory for these instructions, because they must always set WAIT-LOAD. These instructions do not set WAIT-EVENT bits in the CXSTATUS register.

2.5.9. Pipeline

Following execution of a context switch instruction (LW.CSW, LT.CSW, LQ.CSW, WD.CSW, WDLW.CSW, WDLT.CSW, WDLQ.CSW or CSW), the next instruction executes to completion in the current context,

before the context switch is effective. In other words, the context switch — as a result of pipelining — has an architectural "delay slot" exposed to the programmer. This delay slot, and restriction on its usage, is explained below and is generally consistent with similar branch and jump delay slots in the MIPS I ISA.

The delay slot is illustrated below:

	context(i)		context(j)	
inst n	CSW r7		inst m	...
inst n+1	addu r3, r2, r1	—>	inst m+1	addu r7, r6, r3
inst n+2	subu r4, r3, r1		inst m+2	...

In the example, context(i)'s inst n+1 executes to completion. CXPCi stores the address of inst n+2; the address where context(i) resumes when it is later re-activated. After inst n+1 is complete, the next instruction executed is inst m+1 in context(j). Of course, context(i) and context(j) may execute two completely different tasks; or execute the same task on different data (in this case the PC's will also be unrelated).

A number of restrictions apply to the delay slot instruction:

1. No branch or jump may be coded in the delay slot. A context switch changes program flow, like the branch or jump. This restriction is thus similar to the MIPS I restriction that no back-to-back branches or jumps can occur.

2. The register(s) loaded by LW.CSW, LT.CSW, LQ.CSW, WDLW.CSW, WDLT.CSW or WDLQ.CSW cannot be referenced in the delay slot following the load. A similar restriction exists for loads in the MIPS I ISA.

2.5.10. New Context Selection

Following execution of a context switching instruction, the CPU selects the next context for activation from the available pool. The available pool consists of those contexts for which the CXSTATUS register's WAIT-EVENT and WAIT-LOAD fields are clear.

If no context is available, the CPU stalls after executing the context switching instruction and its delay slot. Stall conditions can arise when all contexts initiate long latency processes. For example all contexts might initiate a block transfer within a short period of time such that no transfer has completed when the last context performs its context switch.

The CPU logic required to implement the above next context selection algorithm is pipelined. As a result, the next context selection, in the D-Stage of the pipeline (a critical path), can be very simple. With this approach, the CXSTATUS register sampling used for next context selection will occur several cycles earlier and may not include a newly available context. However, this is not a drawback because event completions for inactive contexts are asynchronous to the current context's program. The LX8380's internal context scheduler (described in the following paragraphs) is pipelined such that if there is currently no active context (all contexts have some wait bit set), it takes two cycles from the time that some context has all of its Wait bits clear, until that context's CXPC value is driven to the instruction RAM.

The LX8380 processor includes internal context scheduling hardware. The scheduler examines the CXSTATUS register of each context to determine which contexts are ready for execution. A context for which all of the WAIT-EVENT and WAIT-LOAD bits are zero may be selected on the next context switch operation. The LX8380's internal context scheduler ignores the THREAD-PRIORITY field of the CXSTATUS register. It selects the next context "fairly". A characteristic of this scheduler is that, if contexts are performing similar types of activities over time, they experience similar selection rates and similar delays in selection when there are multiple contexts ready for execution.

The algorithm employed by the internal scheduler relies on a “window” of ready contexts. The following steps in the algorithm are endlessly repeated:

- Once a window of ready contexts has been chosen, no other contexts are added to this window.
- If a ready context in the window subsequently has one of its Wait bits turned on, that context is removed from the window. Since the window contains only inactive contexts, this can only happen if the currently active context executes a MTCXC to turn on another context’s Wait bit. This is an unusual case because it is expected that MTCXC will only be used during system initialization.
- One-by-one, as context switches are executed, a context from the window is selected for the next context switch. As each context-switch takes effect, the selected context is removed from the window. The selection among the contexts in the window is not architecturally defined and application software should not depend on any particular order. The current implementation selects the highest numbered context in the window, but this may be changed in future implementations.
- When the window is (about to) become empty, a new window is created comprising all of the currently ready contexts. (If there are none, this step repeats until there is at least one ready context.) When a new non-empty window is obtained, the full cycle of this algorithm continues as described above.

Any context that becomes ready will eventually be included in the next new window, and will be selected for execution. Therefore, this algorithm prevents a ready context from being starved out of activation by other contexts. The fairness of this algorithm results from the fact that contexts which become ready more often are dispatched more often while those which become ready less often are dispatched less often.

For applications that require more detailed scheduling, the customer may bypass the standard LX8380 scheduler and supply an application specific design that has access to the same per context information as the standard scheduler. Such a scheduler may also examine other real time information that is outside the province of LX8380 architecture.

The following table lists the ports that the processor supplies for each context, which are directly connected to the standard or application specific scheduler module (the port direction is relative to the processor). An input to the processor must be driven from a register in the scheduler. Likewise, an output from the processor is driven from a register within the processor.

Table 5: Scheduler Ports

Processor Port	Direction	Description
CX_STUSTHWAIT_R[<n>-1:0]	output	asserted when any wait flag is set in CXSTATUS, where <n> is the number of contexts
CX_STUSTHPRIO_R[<n*3>-1:0]	output	THREAD-PRIORITY field from CXSTATUS, where <n> is the number of contexts
CX_THREADACTV_R[<n>-1:0]	output	1 if thread (context) is active, where <n> is the number of contexts
EXT_NEXTCNTXRDY_P_R	input	1 if scheduler’s next context selection is valid
EXT_NEXTCNTX_P_R[2:0]	input	scheduler’s next context selection

Because the scheduler determines the context that the processor will activate on the *next* context switch, it can include register stages in its design to avoid any timing problems. Typically, each processor is connected to its own local context scheduler. However, the use of a single scheduling module, which operates on information from all processors, is not precluded.

It should be noted that the CX_THREADACTV_R signals indicate the current active context at the *end* of the pipeline. Exceptions and mispredicted branches can cause context-switches to be squashed. Furthermore, the WAIT bit values can be set by context switches or MTCXC instructions, and these changes only take effect at the end of the pipeline (after any potential exceptions or branches have been resolved). On the other hand, the EXT_NEXTCNTX_P_R inputs must be used at the *beginning* of the pipeline to select a new active context in case of a potential context switch.

To resolve the discrepancy between the end and beginning of the pipeline, CP0 inhibits a context that is active at any stage of the pipeline from being dispatched for a context switch, regardless of the value of EXT_NEXTCNTX_P_R. In addition, all contexts are inhibited from being dispatched for a context switch while there is an MTCXC instruction at any stage of the pipeline. This will, on rare occasions, cause no valid instructions to be sent down the pipeline, but it eliminates the need for the external scheduler to be aware of the pipeline.

This inhibiting logic also implies that the external scheduler only needs to detect a change in the value of any CX_THREADACTV_R (from zero to one) to determine that a context switch has actually taken place and a new context has been dispatched.

2.5.11. Example Context Switch for Coprocessor Operation

The following example illustrates how an unconditional context switch could be used to allow other contexts to execute while a coprocessor performs a relatively long latency operation on behalf of a context. The example assumes that Coprocessor 2 has been connected to the processor's Coprocessor Interface (CI), which is available as part of Lexra's standard product.

The Coprocessor is assumed to contain a control register ($1) that must contain the context number to which subsequent Coprocessor instructions apply. Another control register ($2) is used to start the Coprocessor operation. When the Coprocessor concludes the operation it signals the processor to clear a specific WAIT-EVENT bit (for the target context) associated with the Coprocessor. This makes the context ready for dispatch. Since several contexts can use Coprocessor 2, before retrieving the results the current context must again be stored to the control register ($1). In addition to the MYCX and CSW instructions, the example uses the MIPS standard MTC2, CTC2, MFC2 instructions for accessing Coprocessor 2.

```
mycx    r1          # get current context number
ctc2    r1, $1      # tell cop2 which context this is
mtc2    ...         # supply other data to cop2
...
csw     r2          # switch, and wait for cop2
ctc2    r3, $2      # kick off cop2 in delay slot
                    # after the context switch,
                    # when the cop2 operation completes
                    # this context is made ready and
                    # eventually gets dispatched here
ctc2    r1, $1      # tell cop2 which context this is
mfc2    ...         # retrieve results
```

2.5.12. Program Access to New Registers

The new registers described in Section 2.5.1. CXPC, CXSTATUS, MOVECX, as well as the general registers of all contexts, are accessible under program control by the active context.

The MOVECX register, which determines the target context for the MTCXC, MFCXC, MTCXG, MFCXG instructions, is loaded by the MTLXC0 instruction and can be read with the MFLXC0 instruction.

The number of the currently executing context can be accessed with the MYCX instruction, which loads it into a general register.

CXPC and CXSTATUS are new Coprocessor 0 registers. These context control registers (ct or cd) can be moved to or from general registers (rt or rd) of the active context using the following instructions:

MTCXC	rt, cd	moves gen reg rt (of the active context) to cd
MFCXC	rd, ct	moves ct to gen reg rd (of the active context)

where,

ct or cd = {CXSTATUS, CXPC}

MOVECX[2:0] designates the context whose ct or cd is to be accessed.

MTCXC and MFCXC should *not* be used to access the CXPC of the currently active context. If ct or cd is the CXPC of the currently active context, the result of MTCXC or MFCXC is undefined.

Two additional instructions permit the general registers (rt or rd) in the active context to be transferred to or from the general registers (gt or gd) in inactive contexts:

MTCXG	rt, gd	moves rt (of the active context) to gd of context MOVECX
MFCXG	rd, gt	moves gt of context MOVECX to rd (of the active context)

This capability is useful in debugging, so that all registers are accessible without execution of a context switch. (The special case of moves within a single context using MTCXG, MFCXG is undetectable by the assembler, though it would normally be performed using a MIPS I instruction.)

Accessing a general register in an inactive context will give unpredictable results if a load is pending to that register.

MTCXC, MFCXC, MTCXG and MFCXG are extensions to the MIPS ISA. They function similarly to the MIPS MTC0 and MFC0 instructions, but the opcodes have different object code assignments to allow the number of Coprocessor 0 registers to be extended. As with MTC0 and MFC0, a Coprocessor Usability Trap is taken in User Mode if CP0 is not designated usable in STATUS (MTCXC, MFCXC, MTCXG, MFCXG are always usable in Kernel Mode.)

2.5.13. Exceptions

The MIPS R3000 exception processing model is unchanged by LX8380, with one difference explained in the next paragraph. Following a program synchronous trap or an interrupt, the PC of the current context is stored in the program-visible EPC register. Exceptions are “precise”, allowing an exception handler to possibly take recovery steps and then resume execution at the PC of the exception. If there is an active context, *no* context switch occurs when an exception (trap or interrupt) is taken. The exception handler executes in the same context that was current at the time the exception was taken. The handler can use the MYCX instruction to

determine its context, if necessary.

LX8380 suppresses exceptions that occur in the delay slot of a context switch. This simplified approach is acceptable in embedded systems. Exception reporting is a useful debug tool during the development process, but is not necessary in production systems. This suppression of exceptions applies to both interrupts and all program synchronous traps. Therefore, instructions which deliberately cause exceptions (BREAK, SYSCALL) should never be coded in the delay slot of a CSW-type instruction. An EJTAG debugger should never attempt to insert an SDBBP in the delay slot, and should also note that single-stepping will execute past the delay slot instruction.

To facilitate system level error detection and reporting, the processor has a special response to the assertion of its IntreqN[7] hardware interrupt input. When this interrupt is asserted, the processor forces context 0 into a ready state by clearing all of the wait flags in context 0's CXSTATUS register. This ensures that there is a context available to service the interrupt. However, the interrupt may be serviced by any other ready context.

All contexts share a common set of Coprocessor 0 registers including the exception processing registers described in Section 2.4, System Control Coprocessor (CP0).

3. RISC Programming Model

This section describes the LX8380 programming model. Section 3.1 summarizes the basic RISC operations supported by the LX8380. 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 LX8380 supports the MIPS R3000 exception processing model, as described in Section 3.4.

The LX8380 supports MIPS I coprocessor operations. The customer can include one or two application-specific coprocessors. The LX8380 includes an optional Coprocessor Interface (CI) that provides a simplified connection between a coprocessor and the internal signals of the LX8380. The CI is described in Section 3.6.

3.1. Summary of Basic RISC Instructions

The LX8380 executes the MIPS I (R2000/R3000) instructions as detailed in the tables below. The LX8380 executes full MIPS I instruction set, excluding the unaligned load and store instructions (LWL, SWL, LWR, SWR) which are executed as no-ops.

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

Additional instructions supported by the LX8380 are described in Section 4, Instruction Extensions; Section 4.2, MAC Instructions; Section 6.3, 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.
cntx::reg	A multi-context *reg* from context *cntx.* Current context if no *cntx*:: prefix.

3.1.1. ALU Instructions

Table 6: 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 7: 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 8: 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 9: 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 10: 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 11: 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 LX8380 based products using the Custom Engine Interface (CEI). The new instructions take one of the following forms illustrated below and use reserved opcodes.

Table 12: 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 LX8380'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 139 for a listing of the CEI signals.

3.3. Simple Memory Management Unit

The LX8380 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 13: 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 LX8380 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 LX8380 implements the MIPS R3000 exception processing model. TLB related exceptions are included only if the LX8380 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 14: 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 LX8380 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 LX8380, 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 14) 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 LX8380 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 LX8380 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 15: 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 LX8380 (not including the BMC, which is implemented as Coprocessor 3). 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 LX8380 includes the optional Coprocessor Interface (CI) allowing the customer to easily interface a coprocessor to the LX8380. 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 5, Coprocessor Interface.

4. Instruction Extensions

4.1. Context Switch and Data Transfer Operations

The table below explains the details of the instructions that are used to cause a context switch, and to transfer data on behalf of a context. The context switching instructions typically set one or more WAIT bits in the context's CXSTATUS register which prevent the context from being reactivated until its program can usefully resume.

Since a thread may wish to wait for notification of up to eight (hardware or software) events, there is a user-mode instruction, POSTCX, which allows another thread to atomically clear any (within this processor) context's WAIT-EVENT bits.

The instruction MYCX allows the program to determine its own context number and, if there are multiple processors in the system, its own processor number. This allows several threads to execute the same program, but to use their context numbers (and/or processor numbers) to access unique memory regions or remote devices.

All of these instructions may be executed in User mode and therefore are *not* subject to any coprocessor usability exceptions.

For all of the instructions which cause a context switch, there is a single instruction delay slot. That is, the instruction immediately following the context-switching instruction is executed in the same context, and that context's CXPC is loaded with the address of the instruction after the delay slot. Immediately after the execution of the delay slot instruction, the newly selected context begins execution at the instruction specified by its CXPC register.

There are restrictions on the type of instruction that can be executed in the delay slot of context switching instructions. These restrictions are detailed in a note following Table 16.

For several of the instructions, the descriptions are nearly identical, differing in only a few items. In order to make it easier for the reader to identify only the differences, these are indicated with underlined text. For an explanation of the conventions employed in the algorithmic descriptions, refer to Section 3.1 on page 21.

Table 16: Context Switching Instructions

Instruction	Syntax and Description
MYCX rD	My Context rD ← { 16'h0000, procNum[7:0], 5'b00000, cntxNum[2:0] } The current context number is placed into rD[2:0]. If there are multiple processors in the system, the number of the processor executing this instruction is placed into rD[15:8]. Otherwise rD[15:8] is zeroed. All other bits of rD are set to zeroes.
POSTCX rS, rT	Post Event to Context cntx ← rT[2:0] temp ← cntx::CXSTATUS[15:8] & rS[31:24] cntx::CXSTATUS[15:8] ← temp Bits rT[2:0] are used as the target context *cntx*. Bits rS[31:24] are logically ANDed with bits 15:8 (the WAIT-EVENT bits) of the CXSTATUS register for context *cntx,* and that context's CXSTATUS register is updated with the result.
CSW rS	Context Switch Unconditional temp ← CXSTATUS[15:8] | rS[31:24] CXSTATUS[15:8] ← temp CXPC ← pc + 8 pc ← next_context::CXPC Bits 15:8 (the WAIT-EVENT bits) from the current context's CXSTATUS register are logically ORed with rS[31:24] and the CXSTATUS register is updated with the result. An unconditional context switch occurs after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated.
LW.CSW rT, offset(rS)	Load Word Uncached with Context Switch temp ← rS[31:0] + { 20 { offset[11] }, offset[11:0] } rT ← Memory[temp] CXSTATUS[3] ← 1'b1 CXPC ← pc + 8 pc ← next_cntx::CXPC The *offset*, in bytes, is a signed 12-bit quantity that must be divisible by 4 (since it occupies only 10 bits of the instruction word). The *offset* is sign extended and added to the contents of rS to form the address *temp*. The word addressed by *temp* is fetched using a split transaction and loaded into rT. The WAIT-LOAD bit is set in this context's CXSTATUS register while the fetch is in progress. An unconditional context switch occurs after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated. If *temp* does not specify an address in uncachable space, the result of the operation is undefined. If *temp* specifies an address in DMEM space, the result of the operation is undefined. If *temp* is not word aligned, an address exception is taken and no context switch occurs.

Instruction	Syntax and Description
LT.CSW rT, offset(rS)	Load TwinWord Uncached with Context Switch temp ← rS[31:0] + { 19 { offset[12] }, offset[12:0] } { rT, rT+1 } ← Memory[temp] CXSTATUS[3] ← 1'b1 CXPC ← pc + 8 pc ← next_cntx::CXPC The *offset*, in bytes, is a signed 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 rS to form the address *temp*. The word addressed by *temp* is fetched using a twinword split transaction, and loaded into rT (which must be an even register). The word addressed by *temp*+4 is loaded into rT+1. The WAIT-LOAD bit is set in this context's CXSTATUS register while the fetches are in progress. An unconditional context switch occurs after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated. If *temp* does not specify an address in uncachable space, the result of the operation is undefined. If *temp* specifies an address in DMEM space, the result of the operation is undefined. If *temp* is not twinword aligned, an address exception is taken and no context switch occurs.
LQ.CSW rT, offset(rS)	Load QuadWord Uncached with Context Switch temp ← rS[31:0] + { 18 { offset[13] }, offset[13:0] } { rT, rT+1, rT+2, rT+3 } ← Memory[temp] CXSTATUS[3] ← 1'b1 CXPC ← pc + 8 pc ← next_cntx::CXPC The *offset*, in bytes, is a signed 14-bit quantity that must be divisible by 16 (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 using a quadword split transaction, and loaded into rT (which must be a register number divisible by four). The word addressed by *temp*+4 is loaded into rT+1. The word addressed by *temp*+8 is loaded into rT+2. The word addressed by *temp*+12 is loaded into rT+3. The WAIT-LOAD bit is set in this context's CXSTATUS register while the fetches are in progress. An unconditional context switch occurs after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated. If *temp* does not specify an address in uncachable space, the result of the operation is undefined. If *temp* specifies an address in DMEM space, the result of the operation is undefined. If *temp* is not quadword aligned, an address exception is taken and no context switch occurs.

Instruction	Syntax and Description
WD[.CSW] rS, rT, devID	Write Descriptor addr ← { system_contant[23:0], devID[4:0], 3'b000 } Memory[addr] ← { rS, rT } if (.CSW) temp ← CXSTATUS[15:8] | rS[31:24] CXSTATUS[15:8] ← temp CXPC ← pc + 8 pc ← next_cntx::CXPC A 64-bit descriptor is formed, with the contents of rS in bits 63:32 and the contents of rT in bits 31:0. If the optional.CSW extension is specified, then bits 63:56 of the descriptor are logically OR-ed with the WAIT-EVENT bits of this context's CXSTATUS register, which is updated with the result. The processor constructs a system bus address with bits 31:8 set to a system-specific constant, bits 7:3 set to the value of the 5-bit *devID* field, and bits 2:0 all zeroes. A system bus operation is performed to write bits 63:0 of the descriptor to the device. If the optional.CSW extension is specified, the processor performs a context switch after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated.

Instruction	Syntax and Description
WDLW.CSW rD, rS, rT, devID	Write Descriptor with Load Word Uncached and Context Switch addr ← { system_contant[23:0], devID[4:0], 3'b000 } Memory[addr] ← { rS, rT } rD ← Memory[addr] CXSTATUS[3] ← 1'b1 CXPC ← pc + 8 pc ← next_cntx::CXPC A 64-bit descriptor is formed, with the contents of rS in bits 63:32 and the contents of rT in bits 31:0. The WAIT-LOAD bit of this context's CXSTATUS register is set. The processor constructs a system bus address with bits 31:8 set to a system-specific constant, bits 7:3 set to the value of the 5-bit *devID* field, and bits 2:0 all zeroes. A system bus operation is performed to write bits 63:0 of the descriptor to the device, also requesting an uncached split transaction read word response. The processor performs a context switch after the execution of this instruction's delay slot. When the processor receives the corresponding read word response from the system bus, it is loaded into register rD of the originating context's general purpose register file and that context's WAIT-LOAD flag is cleared.
WDLT.CSW rD, rS, rT, devID	Write Descriptor with Load Twinword Uncached and Context Switch addr ← { system_contant[23:0], devID[4:0], 3'b000 } Memory[addr] ← { rS, rT } { rD, rD+1 } ← Memory[addr] CXSTATU[3] ← 1'b1 CXPC ← pc + 8 pc ← next_cntx::CXPC A 64-bit descriptor is formed, with the contents of rS in bits 63:32 and the contents of rT in bits 31:0. The WAIT-LOAD bit of this context's CXSTATUS register is set. The processor constructs a system bus address with bits 31:8 set to a system-specific constant, bits 7:3 set to the value of the 5-bit *deviceID* field, and bits 2:0 all zeroes. A system bus operation is performed to write bits 63:0 of the descriptor to the device, also requesting an uncached split transaction read twinword response. The processor performs a context switch after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated. When the processor receives the corresponding read twinword response from the system bus, the first returned word is loaded into register rD (which must specify an even register), and the second returned word is loaded into rD+1 of the originating context's general purpose register file, and that context's WAIT-LOAD flag is cleared.

Instruction	Syntax and Description
WDLQ.CSW rD, rS, rT, devID	Write Descriptor with Load Quadword Uncached and Context Switch addr ← { system_contant[23:0], devID[4:0], 3'b000 } Memory[addr] ← { rS, rT } { rD, rD+1, rD+2, rD+3 } ← Memory[addr] CXSTATUS[3] ← 1'b1 CXPC ← pc + 8 pc ← next_cntx::CXPC A 64-bit descriptor is formed, with the contents of rS in bits 63:32 and the contents of rT in bits 31:0. The WAIT-LOAD bit of this context's CXSTATUS register is set. The processor constructs a system bus address with bits 31:8 set to a system-specific constant, bits 7:3 set to the value of the 5-bit *devID* field, and bits 2:0 all zeroes. A system bus operation is performed to write bits 63:0 of the descriptor to the device, also requesting an uncached split transaction read quadword response. The processor performs a context switch after the execution of this instruction's delay slot. The context scheduler determines the next context that is activated. When the processor receives the corresponding read quadword response from the system bus, the first returned word is loaded into register rD (which must specify a register number divisible by four), the second returned word is loaded into rD+1, the third returned word is loaded into rD+2, and the fourth returned word is loaded into rD+3 of the originating context's general purpose register file, and that context's WAIT-LOAD flag is cleared.

Nomenclature: rS, rT, rD = r0 - r31
base = r0 - r31

Notes: The delay slot of the CSW, LW.CSW, LT.CSW, LQ.CSW, WD.CSW, WDLW.CSW, WDLT.CSW and WDLQ.CSW instructions may not contain a branch, jump or MTCXC instruction.

The delay slot of the WDLW.CSW, WDLT.CSW WDLQ.CSW, WDLW.CSW, WDLT.CSWand WDLQ.CSW instructions may not access to any register loaded by the instruction

4.2. Bit Field Processing Operations

Table 17 explains the details of the instructions used to manipulate bit fields.

As shown in Figure 4, for several of these instructions a width and insert offset specify a subfield of a 32-bit register that is to be used as a target of the instruction. For the EXTIV and INSV paired instructions (or EXTII and INSI), the extract offset and width can specify a (maximally 32-bit) subfield which straddles the boundary of two source registers or is completely contained in either one of two potential source registers. Figure 4, Insert and Extract Operations (Straddle Case), illustrates the straddle case.

It is worth noting that the standard MIPS instruction set includes Branch On Equal, and Branch On Not Equal instructions. Therefore, the Extract instruction can be used to select a field that is tested by a conditional branch, and no explicit Test instruction is necessary.

For several of the instructions, the descriptions are nearly identical, differing in only a few items. In order to make it easier for the reader to identify only the differences, these are indicated with underlined text.

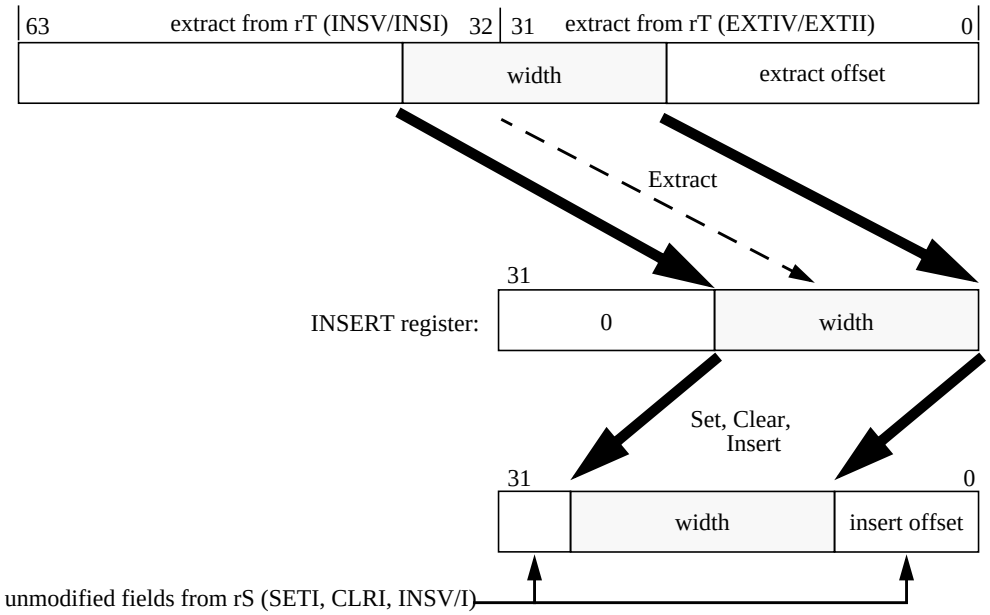


Figure 4: Insert and Extract Operations (Straddle Case)

Table 17: Bit Field Processing Instructions

Instruction	Syntax and Description
SETI rT, rS, width, offset	Set Bits Immediate rD[width+offset-1:offset] ← width { 1'b1 } The *offset* is a value p in the range 0-31. The *width* is a value m in the range 1-32 (which is encoded in the instruction as a 5-bit value modulo 32 — that is, the value 32 is encoded as zero). The bits rT[m+p-1:p] are set to ones. The remaining bits of rT are copied from the corresponding bits of rS. If m+p is greater than 32, the results are unpredictable.
CLRI rT, rS, width, offset	Clear Bits Immediate rD[width+offset-1:offset] ← width { 1'b0 } The *offset* is a value p in the range 0-31. The *width* is a value m in the range 1-32 (which is encoded in the instruction as a 5-bit value modulo 32 — that is, the value 32 is encoded as zero). The bits rT[m+p-1:p] are set to zeroes. The remaining bits of rT are copied from the corresponding bits of rS. If m+p is greater than 32, the results are unpredictable.

Instruction	Syntax and Description

EXTIV rD, rS, rT	Extract Bits for Insertion Variable
	
	INSERT[47:32] ← rS[15:0]
	xoffset ← rS[15:10]
	width ← rS[9:5]
	
	if (xoffset<32)
	  if ((xoffset+width-1)<32)
	  temp[width-1:0] ← rT[width+xoffset-1:offset]
	  temp[31:width] ← (32-width) { 1'b0 }
	  else
	  temp[31-xoffset:0] ← rT[31:xoffset]
	  temp[31:32-xoffset] ← (32-width) { 1'b0 }
	else
	  temp[31:0] ← 32'h0000_0000 }
	
	rD ← temp
	INSERT[31:0] ← temp
	
	The bits rS[15:10] are decoded as an extraction offset n in the range 0-63. The bits rS[9:5] are decoded as a width m in the range 1-32 modulo 32. The bits rS[4:0] are decoded as an insertion offset p in the range 0-31. These *parameter* fields of rS are saved in the implied register INSERT. The remaining bits of rS are ignored. Considering rT to contain the least significant 32 bits of the extraction source, a 32-bit intermediate extraction value *temp* is generated as follows:
	1) if n<32 and (n+m-1) < 32, (least significant word only) the bits rT[m+n-1:n] are copied into *temp*[m-1:0] and the remaining bits of *temp* are set to zeroes.
	2) if n<32 and (n+m-1) > 31, (straddle two words) the bits rT[31:n] are copied into *temp*[31-n:0] and the remaining bits of *temp* are set to zeroes.
	3) if n>31, (most significant word only) *temp*[31:0] is set to all zeroes.
	
	The *temp* value is stored in rD and also saved in the implied register INSERT.
	If m+n is greater than 64, the results of this instruction, and a subsequent INSV instruction are unpredictable.

Instruction	Syntax and Description

INSV rD, rS, rT	Insert Bits Variable
	
	xoffset ← INSERT[47:42]
	width ← INSERT[41:37]
	ioffset ← INSERT[36:32]
	temp ← INSERT[31:0]
	
	if (xoffset<32)
	  if ((xoffset+width-1)<32)
	  result[31:0] ←temp[31:0]
	  else
	  result[31-xoffset:0] ← temp[31-xoffset:0]
	  result[width-1:32-xoffset] ← rT[xoffset+width-33:0]
	  if (width<32)
	  result[31:width] = (32-width) { 1'b0 }
	else
	  result[width-1:0] ← rT[xoffset+width-33:xoffset-32]
	  result[31:width] ← (32-width) { 1'b0 }
	
	rD[width+ioffset-1:ioffset] ← result[width-1:0]
	
	This instruction must be coded as the next sequential instruction in the program sequence after an EXTIV. Otherwise, its results are unpredictable.
	All exceptions are inhibited for the execution of this instruction. This includes hardware interrupts, debug exceptions and address exceptions.
	The parameter fields m, n, p and the intermediate extraction value *temp* are taken from the implied register INSERT, as described for EXTIV. Considering rT to contain the most significant 32 bits of the extraction source, the final extracted value *result* is generated as follows:
	1) if n<32 and (n+m-1) < 32, the bits *temp*[31:0] are copied into *result*[31:0].
	2) if n<32 and (n+m-1) > 31, the bits *temp*[31-n:0] are copied into *result*[31-n:0]. The bits rT[n+m-33:0] are copied into *result*[m-1:32-n]. The remaining bits of *result* are set to zeroes.
	3) if n>31, the bits rT[n+m-33:n-32] are copied into *result*[m-1:0]. The remaining bits of *result* are set to zeroes.
	
	The bits from *result*[m-1:0] are copied into rD[m+p-1:p]. The remaining bits of rD are copied from the corresponding bits of rS. If m+n is greater than 64, or if m+p is greater than 32, the results are unpredictable.

Instruction	Syntax and Description

EXTII rD, rT, width, xoffset	Extract Bits for Insertion Immediate
	
	INSERT[47:42] ← xoffset
	INSERT[41:37] ← width
	
	if (xoffset<32)
	  if ((xoffset+width-1)<32)
	  temp[width-1:0] ← rT[width+xoffset-1:offset]
	  temp[31:width] ← (32-width) { 1'b0 }
	  else
	  temp[31-xoffset:0] ← rT[31:xoffset]
	  temp[31:32-xoffset] ← (32-width) { 1'b0 }
	else
	  temp[31:0] ← 32'h0000_0000 }
	
	rD ← temp
	INSERT[31:0] ← temp
	
	The extract offset *xoffset* is a value n in the range 0-31. The *width* is a value m in the range 1-32 (which is encoded in the instruction as a 5-bit value modulo 32 — that is, the value 32 is encoded as zero). These *parameter* fields are saved in the implied register INSERT. Considering rT to contain the least significant 32 bits of the extraction source, a 32-bit intermediate extraction value *temp* is generated as follows:
	1) if (n+m-1) < 32, (least significant word only) the bits rT[m+n-1:n] are copied into *temp*[m-1:0] and the remaining bits of *temp* are set to zeroes.
	2) if (n+m-1) > 31, (straddle two words) the bits rT[31:n] are copied into *temp*[31-n:0] and the remaining bits of *temp* are set to zeroes.
	
	The *temp* value is stored in rD and also saved in the implied register INSERT.

Instruction	Syntax and Description

INSI rD, rS, rT, ioffset	Insert Bits Immediate
	
	xoffset ← INSERT[47:42]
	width ← INSERT[41:37]
	temp ← INSERT[31:0]
	
	if (xoffset<32)
	  if ((xoffset+width-1)<32)
	  result[31:0] ←temp[31:0]
	  else
	  result[31-xoffset:0] ← temp[31-xoffset:0]
	  result[width-1:32-xoffset] ← rT[xoffset+width-33:0]
	  if (width<32)
	  result[31:width] = (32-width) { 1'b0 }
	else
	  result[width-1:0] ← rT[xoffset+width-33:xoffset-32]
	  result[31:width] ← (32-width) { 1'b0 }
	
	rD[width+ioffset-1:ioffset] ← result[width-1:0]
	
	This instruction must be coded as the next sequential instruction in the program sequence after an EXTII. Otherwise, its results are unpredictable.
	All exceptions are inhibited for the execution of this instruction. This includes hardware interrupts, debug exceptions and address exceptions.
	The parameter fields m, n and the intermediate extraction value *temp* are taken from the implied register INSERT, as described for EXTII. The insert offset *ioffset* is a value p in the range 0-31. Considering rT to contain the most significant 32 bits of the extraction source, the final extracted value *result* is generated as follows:
	1) if (n+m-1) < 32, the bits *temp*[31:0] are copied into *result*[31:0].
	2) if (n+m-1) > 31, the bits *temp*[31-n:0] are copied into *result*[31-n:0]. The bits rT[n+m-33:0] are copied into *result*[m-1:32-n]. The remaining bits of *result* are set to zeroes.
	
	The bits from *result*[m-1:0] are copied into rD[m+p-1:p]. The remaining bits of rD are copied from the corresponding bits of rS. If m+p is greater than 32, the results are unpredictable.

Instruction	Syntax and Description
HASH rD, rS, keysize	Hash to Key rD ← Hash (rS[keysize-1:0], keysize) The 5-bit keysize is a value k in the range 4-24. If k is outside this range, the results are unpredictable. The 32 *source* bits contained in rS are hashed to form a *key* of k bits. The *key* is stored in rD[k-1:0]. The remaining bits of rD are zeroed. For a given keysize, each bit of the *key* is formed as the logical XOR of a subset of the *source* bits. For any keysize these subsets are mutually exclusive and exhaustive. That is, each source bit is included in the XOR function of one and only one of the key bits. The composition of the XOR subsets for each keysize is indicated in Table 18, Hash Instruction Key Bit Definition.
MSB rD, rS, rT	Most Significant Bit Encode temp ← rS & rT msb ← 33 repeat msb ← msb - 1 until (temp[msb-1] || (msb=0)) rD[31:6] ← 0 rD[5:0] ← msb The 32-bit *temp* is computed as the logical AND of rS with rT. The 6-bit *result* indicates the most significant bit that is set in *temp* according to the following table (where “x” means don’t care): *temp* = `00000000 00000000 00000000 00000000` : *result*= 0 *temp* = `00000000 00000000 00000000 00000001` : *result*= 1 *temp* = `00000000 00000000 00000000 0000001x` : *result*= 2 *temp* = `00000000 00000000 00000000 000001xx` : *result*= 3 etc. *temp* = `1xxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx` : *result*= 32 The *result* is stored in rD[5:0]. The remaining bits of rD are zeroed.

Instruction	Syntax and Description
JOR rS, rT	Jump to Offset Register offset ← rS[15:3] | rT[12:0] target[31:16] ← rS[31:16] target[15:3] ← offset target[2:0] ← 3'b000 pc ← target The 13-bit jump *offset* is computed as the logical OR of rT[12:0] with rS[15:3] The 32-bit *target* address is computed as follows: *target* [31:16] = rS[31:16] *target* [15:3] = *offset* *target* [2:0] = zeroes. The other bits of rT and rS are ignored. The program unconditionally jumps to the *target* address with a delay of one instruction just like the JR instruction. Handling of the delay slot instruction for exceptions is the same as for the JR instruction.

Nomenclature: rT, rS, rD = r0 - r31

Notes: For EXTIV, specifying r0 for rS implies (insert / extract) offsets of 0 and a width of 32.

INSV (INSI) must be coded as the next sequential instruction following EXTIV (EXTII). There is only one INSERT register in the processor (not one per context) which only exists to pass information from EXTIV(EXTII) to INSV(INSI). The processor inhibits exceptions for INSV(INSI) to ensure that if the EXTIV(EXTII) instruction completes, the immediately subsequent INSV(INSI) will also complete.

For EXTII the extract offset may not be > 32 (but straddle is allowed) due to format constraints. This should NOT be a problem since the immediate is known at compile time. If an offset > 32 were needed, the next most significant register could be used for rT and the offset reduced by 32.

The EXTIV and INSV pair of instructions are intended to allow numerous non-contiguous fields in a packet to be compacted into a single contiguous key. Even if the alignment of the packet in a set of registers is not known until run time, a sequence of 3 instructions per field can be used to accomplish this compaction.

In the example in Figure 5, packet data is loaded into source registers s1, s2, and s3 and fields F1 and F2 are to be compacted into destination register d1. However, it is not known until run time which of four byte alignment cases of the packet is valid. At run time, r1 is loaded with a value corresponding to the alignment. Specifically, the value needed in bits 15:10 of r1 is the two's complement of the alignment in bits. A single instruction (ori r1, r0, (-n<<10)) loads the proper value for any of the cases.

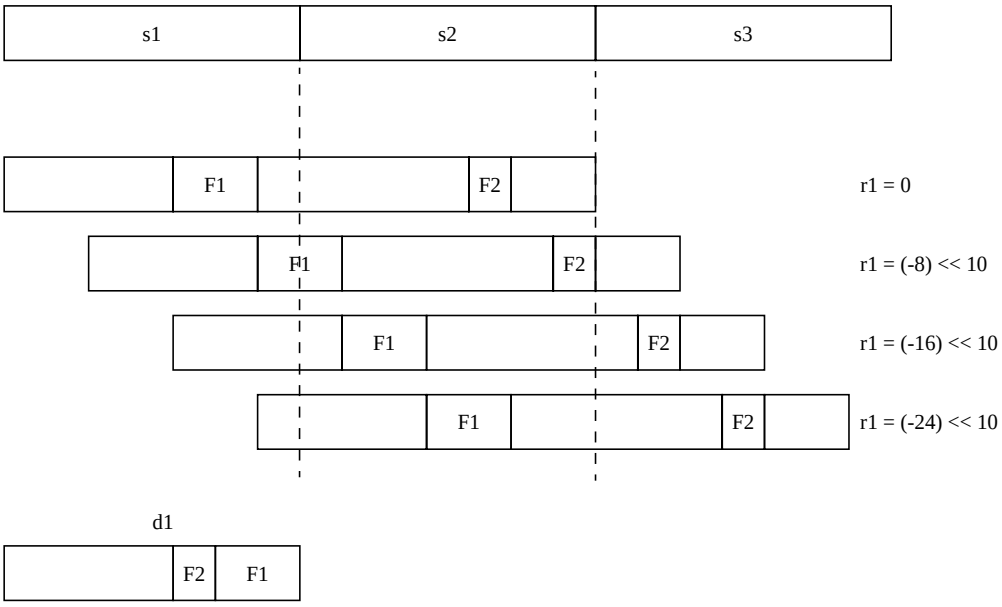


Figure 5: Packet Field Compaction with Variable Alignment

The following code sequence assumes that r1 has been initialized as needed according to the case in question. As shown, a common code path is used regardless of the alignment. Note that r0 is a 0 source and don't care destination.

```
# r1 contains the value to be subtracted
# from the 6-bit default extraction offsets.

addiu       r2, r1, (F1_OFFE<<10 + F1_WID<<5 + F1_OFFI)
extiv       r0, r2, s2          # F1 is from s1 and/or s2
insv        d1, r0, s1          # insert F1 into d1
addiu       r2, r1, (F2_OFFE<<10 + F2_WID<<5 + F2_OFFI)
extiv       r0, r2, s3          # F2 is from s2 and/or s3
insv        d1, d1, s2          # merge F2 into d1
...more fields handled the same way
```

The above example shows how the packet alignment is handled with a value held in a single register, placed in the appropriate bit position, so that it can be subtracted from the otherwise fixed extraction offsets. The widths and insertion offsets are invariant. This paradigm works provided that two conditions are met:

1) The variability in alignment never causes a field to straddle different pairs of source registers. A sufficient condition is if the extracted field does not cross a word boundary in the nominal case (in other words, the default extract offset is greater than 31.)

2) The insertion width and alignment never cause a field to straddle a word boundary in the destination key. This problem can be minimized by reordering the fields in the destination key, but in the worst case, a field to may be split into two parts to avoid the issue.

If necessary, both of these restrictions can always be satisfied by splitting some source fields into two fields.

Table 18: Hash Instruction Key Bit Definition

Keysize	KeyBit	Source Bits Included in XOR to form Key Bit
4	3	28 24 20 16 12 8 4 0
	2	29 25 21 17 13 9 5 1
	1	30 26 22 18 14 10 6 2
	0	31 27 23 19 15 11 7 3

Keysize	KeyBit	Source Bits
5	4	26 25 16 9 3 0
	3	28 24 20 12 8 4
	2	29 21 17 13 5 1
	1	30 22 18 14 10 6 2
	0	31 27 23 19 15 11 7
6	5	26 24 18 10 9 1
	4	25 19 16 11 3 0
	3	28 20 12 8 4
	2	29 21 17 13 5
	1	30 22 14 6 2
	0	31 27 23 15 7
7	6	25 16 9 1
	5	26 24 18 10
	4	19 11 3 0
	3	28 20 12 8 4
	2	29 21 17 13 5
	1	30 22 14 6 2
	0	31 27 23 15 7
8	7	24 16 8 0
	6	25 17 9 1
	5	26 18 10 2
	4	27 19 11 3
	3	28 20 12 4
	2	29 21 13 5
	1	30 22 14 6
	0	31 23 15 7

Keysize	KeyBit	Source Bits
9	8	26 16 9
	7	24 8 0
	6	25 17 1
	5	18 10 2
	4	27 19 11 3
	3	28 20 12 4
	2	29 21 13 5
	1	30 22 14 6
	0	31 23 15 7
10	9	26 13 9
	8	20 16 3
	7	24 8 0
	6	25 17 1
	5	18 10 2
	4	27 19 11
	3	28 12 4
	2	29 21 5
	1	30 22 14 6
	0	31 23 15 7
11	10	30 7
	9	26 13 9
	8	20 16 3
	7	24 8 0
	6	25 17 1
	5	18 10 2
	4	27 19 11
	3	28 12 4
	2	29 21 5
	1	22 14 6
	0	31 23 15
12	11	7 3
	10	30 26
	9	13 9
	8	20 16
	7	24 8 0
	6	25 17 1
	5	18 10 2
	4	27 19 11
	3	28 12 4
	2	29 21 5
	1	22 14 6
	0	31 23 15
13	12	20 13
	11	7 3
	10	30 26
	9	25 9
	8	16 0
	7	24 8
	6	17 1
	5	18 10 2
	4	27 19 11
	3	28 12 4
	2	29 21 5
	1	22 14 6
	0	31 23 15
14	13	30 13
	12	20 7
	11	19 3
	10	26 10
	9	25 9
	8	16 0
	7	24 8
	6	17 1
	5	18 2
	4	27 11
	3	28 12 4
	2	29 21 5
	1	22 14 6
	0	31 23 15

Keysize	KeyBit	Source Bits	Keysize	KeyBit	Source Bits	Keysize	KeyBit	Source Bits
	14	29 13		15	20 4		16	4
	13	30 4		14	29 13		15	20
	12	20 7		13	30 14		14	29 13
	11	19 3		12	23 7		13	30 14
	10	26 10		11	19 3		12	23 7
	9	25 9		10	26 10		11	19 3
	8	16 0		9	25 9		10	26 10
	7	24 8		8	16 0		9	25 9
15	6	17 1	**16**	7	24 8	**17**	8	16 0
	5	18 2		6	17 1		7	24 8
	4	27 11		5	18 2		6	17 1
	3	28 12		4	27 11		5	18 2
	2	21 5		3	28 12		4	27 11
	1	22 14 6		2	21 5		3	28 12
	0	31 23 15		1	22 6		2	21 5
				0	31 15		1	22 6
							0	31 15
	17	29		18	14		19	23
	16	4		17	29		18	14
	15	20		16	4		17	29
	14	13		15	20		16	4
	13	30 14		14	13		15	20
	12	23 7		13	30		14	13
	11	19 3		12	23 7		13	30
	10	26 10		11	19 3		12	7
	9	25 9		10	26 10		11	19 3
	8	16 0		9	25 9		10	26 10
18	7	24 8	**19**	8	16 0	**20**	9	25 9
	6	17 1		7	24 8		8	16 0
	5	18 2		6	17 1		7	24 8
	4	27 11		5	18 2		6	17 1
	3	28 12		4	27 11		5	18 2
	2	21 5		3	28 12		4	27 11
	1	22 6		2	21 5		3	28 12
	0	31 15		1	22 6		2	21 5
				0	31 15		1	22 6
							0	31 15
	20	3		21	26		22	9
	19	23		20	3		21	26
	18	14		19	23		20	3
	17	29		18	14		19	23
	16	4		17	29		18	14
	15	20		16	4		17	29
	14	13		15	20		16	4
	13	30		14	13		15	20
	12	7		13	30		14	13
	11	19		12	7		13	30
	10	26 10		11	19		12	7
21	9	25 9	**22**	10	10	**23**	11	19
	8	16 0		9	25 9		10	10
	7	24 8		8	16 0		9	25
	6	17 1		7	24 8		8	16 0
	5	18 2		6	17 1		7	24 8
	4	27 11		5	18 2		6	17 1
	3	28 12		4	27 11		5	18 2
	2	21 5		3	28 12		4	27 11
	1	22 6		2	21 5		3	28 12
	0	31 15		1	22 6		2	21 5
				0	31 15		1	22 6
							0	31 15

Keysize	KeyBit	Source Bits
24	23	16
	22	9
	21	26
	20	3
	19	23
	18	14
	17	29
	16	4
	15	20
	14	13
	13	30
	12	7
	11	19
	10	10
	9	25
	8	0
	7	24 8
	6	17 1
	5	18 2
	4	27 11
	3	28 12
	2	21 5
	1	22 6
	0	31 15

4.3. Cross Context Access Operations

Table 19 explains the details of instructions that are used to access the general registers or the context control registers of another context. For the control registers, it is also possible for a thread to access its own CXSTATUS register.

The target context for all of these instructions is specified in a new Lexra Coprocessor 0 register, called MOVECX. That register is itself accessed with MTLXC0 and MFLXC0 variants of the MIPS standard MTC0 and MFC0 instructions. These new instructions are used to access Lexra defined Coprocessor 0 registers that are not in the standard MIPS Coprocessor 0 space. The encoding of these instructions, which use the COP0 major opcode, is described in Section 4.5.

It is expected that these instructions will only be used in kernel mode. Therefore, they are all subject to the Coprocessor Unusable exception for Coprocessor 0 as are the MTLXC0 and MFLXC0 instructions.

Table 19: Cross Context Access Instructions

Instruction	Syntax and Description
MFCXG rD, gT	Move From Context General Register cntx ← MOVECX[2:0] rD ← cntx::gT Bits MOVECX[2:0] are used to determine the source context *cntx*. The contents of general register gT in context *cntx* are loaded into the current context's general register rD
MTCXG rT, gD	Move To Context General Register cntx ← MOVECX[2:0] cntx::gD ← rT Bits MOVECX[2:0] are used to determine the target context *cntx*. The general register gD in context *cntx* is loaded from the contents of the current context's general register rT.
MFCXC rD, cT	Move From Context Control Register cntx ← MOVECX[2:0] rD ← cntx::cT Bits MOVECX[2:0] are used to determine the source context *cntx*. The contents of control register cT in context *cntx* are loaded into the current context's general register rD. If a MFCXC instruction that reads CXSTATUS of a target context is executed as the first instruction immediately following a POSTCX to that context, it is unpredictable whether MFCXC returns the new or old value of CXSTATUS.
MTCXC rT, cD	Move To Context Control Register cntx ← MOVECX[2:0] cntx::cD ← rT Bits MOVECX[2:0] are used to determine the target context *cntx*. The control register cD in context *cntx* is loaded from the contents of the current context's general register rT.

Nomenclature:

rT, rD, gT, gD	=	r0 - r31
cD, cT	=	CXSTATUS, CXPC

Notes: Execution of MTCXC rT, CXPC with *MOVECX*= current context (attempt to change the currently executing context's CXPC) results in unpredictable operation.

To examine its own CXSTATUS register a thread can execute this sequence:

```
MYCX      r1
MTLXC0    r1, MOVECX
MFCXC     r2, CXSTATUS
```

4.4. Checksum Addition

Table 20 explains the instruction that may be used to calculate a checksum for an Internet Protocol Header using 16-bit ones complement addition.

Table 20: Checksum Addition Instructions

Instruction	Syntax and Description
ACS2 rD, rS, rT	Dual Add for Checksum templo[16:0] ← { 1'b0, rS[15:0] } + { 1'b0, rT[15:0] } templo[15:0] ← templo[15:0] + { 15'h000, templo[16] } temphi[16:0] ← { 1'b0, rS[31:16] } + { 1'b0, rT[31:16] } temphi[15:0] ← temphi[15:0] + { 15'h000, temphi[16] } rD ← { temphi[15:0], templo[15:0] } Dual 16-bit ones complement addition. Considering all quantities as unsigned 16-bit integers, add the contents of rS[15:00] to rT[15:00] and, independently add the contents of rS[31:16] to rT[31:16]. For each independent addition, if there is a carry out of the most significant bit of its result, add one to that result to form its final result. The final results of the two additions are placed in rD[15:00] and rD[31:16].

Notes: In ones complement arithmetic there are two representations of zero: 0x0000 (+0) and 0xffff (-0). Addition of non-zero quantities can never result in +0, only -0. Addition of -0 to either +0 or -0 results in -0.

This instruction can be used to generate or check the 16-bit checksum used in internet packets. Without regard to halfword alignment, all of the 32-bit words to be included are incrementally added using ACS2. A final 16-bit shift and one more ACS2 instruction is used to "fold" the checksum into 16 bits:

```
la      r1, PACKETADDR      # get packet address
lw      r2, 0(r1)           # get many words
lw      r3, 4(r1)
lw      r4, 8(r1)
lw      r5, 12(r1)
lw      r6, 16(r1)
lw      r7, 20(r1)
...
acs2    r2, r2, r3          # add them together
acs2    r2, r2, r4
acs2    r2, r2, r5
acs2    r2, r2, r6
acs2    r2, r2, r7
...
srl     r3, r2, 16          # fold over accumulator
acs2    r2, r2, r3          # r2[15:0] has the answer
```

4.5. LX8380 Instruction Summary

Table 21: Instruction Summary

Instruction		Description
Context Control Operations and Data Transfers		
MYCX	rD	read My Context
POSTCX	rS, rT	Post event to a Context
CSW	rS	Context Switch
LTW	rT, disp(base)	Load Twinword
LW.CSW	rT, disp(base)	Load Word Uncached with Context Switch
LT.CSW	rT, disp(base)	Load Twinword Uncached with Context Switch
LQ.CSW	rT, disp(base)	Load Quadword Uncached with Context Switch
WD.[CSW]	rS, rT, devID	Write Descriptor to Device [with Context Switch]
WDLW.CSW	rD, rS, rT, devID	Write Descriptor to Device and Load Word/Twinword/Quadword Uncached with Context Switch
WDLT.CSW	rD, rS, rT, devID	
WDLQ.CSW	rD, rS, rT, devID	
Bit Field Operations		
SETI	rT, rS, width, offset	Set Subfield to Ones
CLRI	rT, rS, width, offset	Clear Subfield to Zeroes
EXTIV	rD, rS, rT	Extract Subfield and prepare for Insertion Variable
INSV	rD, rS, rT	Insert Extracted Subfield Variable
EXTII	rD, rT, width, offset	Extract Subfield and prepare for Insertion Immediate
INSI	rD, rS, rT, offset	Insert Extracted Subfield Variable Immediate
ACS2	rD, rS, rT	Dual 16-bit Ones Complement Add for Checksum
HASH	rD, rS, keysize	Hash data to a key
MSB	rD, rS, rT	Find Most Significant Bit
JOR	rS, rT	Jump to Offset Register
Cross-Context Access Operations		
MFCXG	rD, gT	Move from a Context gpr
MTCXG	rT, gD	Move to a Context gpr
MFCXC	rD, cT	Move from a Context control register
MTCXC	rT, cD	Move to a Context control register

5. Coprocessor Interface

The LX8380 processor provides Coprocessor Interfaces (CIs) for the attachment of application-specific coprocessors. This section provides a description of these access points.

5.1. 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).

5.2. 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 LX8380 CI.

Table 22: Coprocessor Interface Signals

Signal	I/O	Description
Czcondin	input	Branch flag.
Czrd_addr[4:0]	output	Read address.
Czrd_cntx_[2:0]	output	Cop read context number.
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_cntx[2:0]	output	Cop write context number.
Czwr_gen	output	General register write command.

Signal	I/O	Description
Czwr_con	output	Control register write command.
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.

5.3. 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 6 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. (The Czwr_cntx_W[2:0] output is not shown in the diagram. The transitions on this signal correspond to transitions on Czwr_addr_W[2:0].)

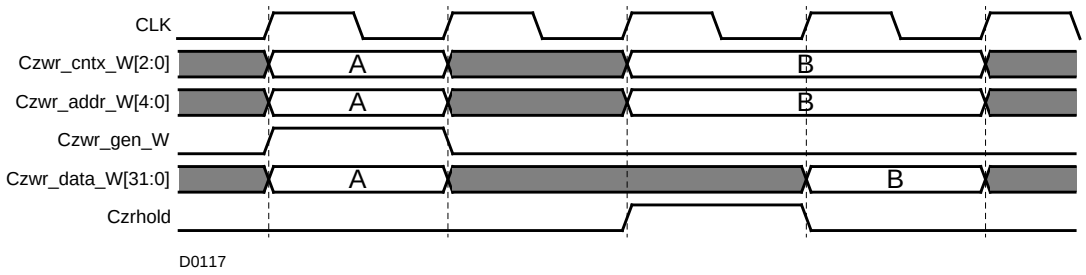


Figure 6: Coprocessor Write

5.4. 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 7 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 8 and Figure 9 for examples of instruction cancellations).

In the example of Figure 7, 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. (The

Czrd_cntx_S[2:0] output is not shown in the diagram. The transitions on this signal correspond to transitions on Czrd_addr_S[2:0].)

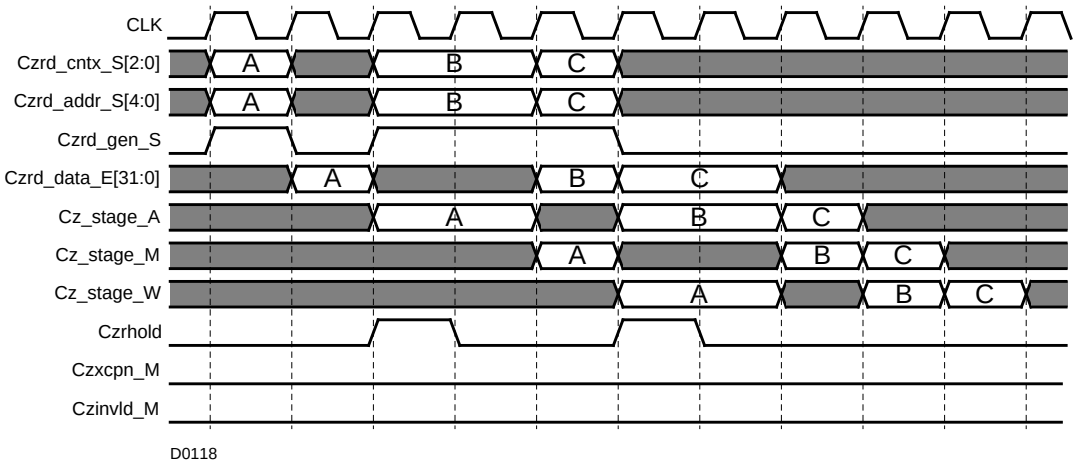


Figure 7: Coprocessor Read

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
```

5.5. 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
```

5.5.1. 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 6.

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 7.

5.5.2. 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 8 illustrates the occurrence of an exception while a coprocessor instruction is at the S, E or A stages.

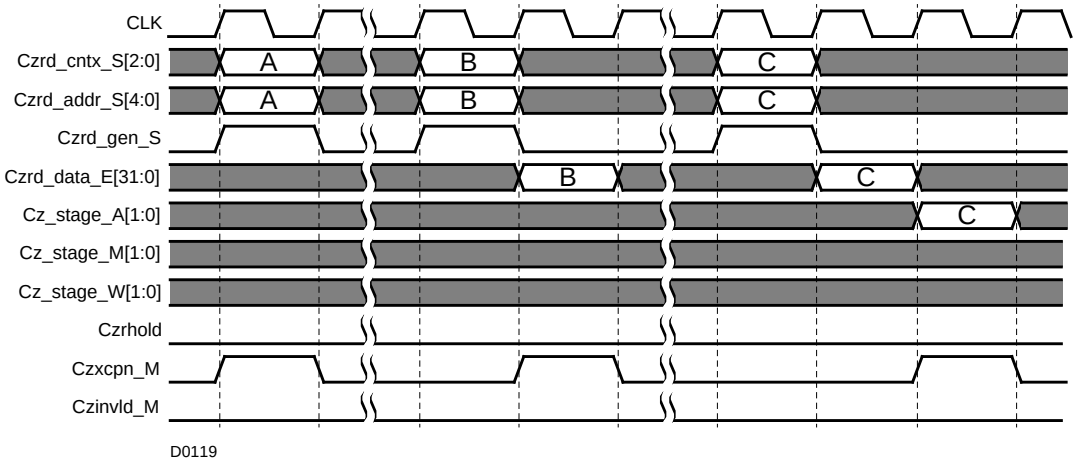


Figure 8: Exception During Coprocessor Read

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 9.

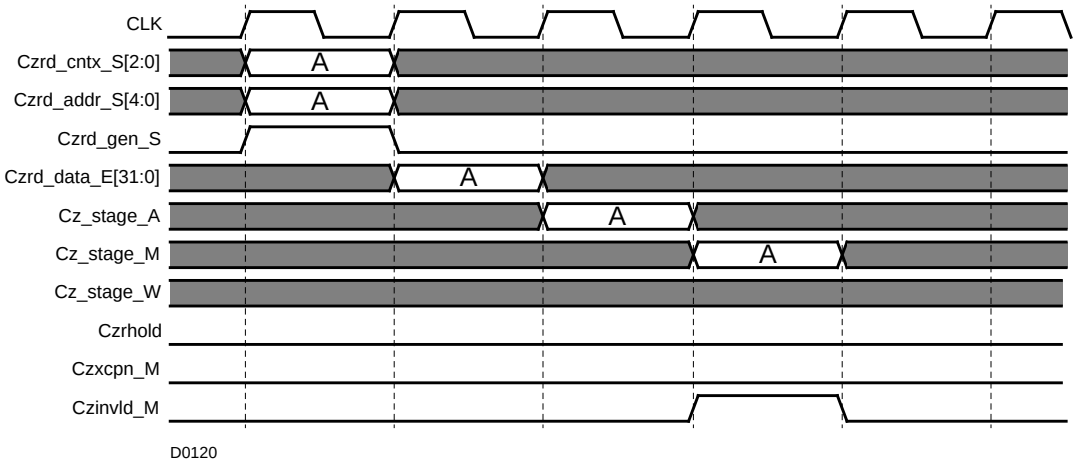


Figure 9: Invalidation of Coprocessor Read

6. Local Memory

6.1. Local Memory Overview

This section describes how memories are configured and connected to the LX8380 using the Local Memory Interfaces (LMIs). This section provides a brief summary of the conventions and supported memories. Section 6.2 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 LX8380 LMIs. All of the signals between the LMIs and RAMs are automatically configured by *lconfig*, the LX8380 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 LX8380 processor's local instruction and data busses. The LMIs also provide the pathways from the processor to the system bus. The LX8380 includes an LMI for each of the local memory types. The sizes of the RAMs are customer selectable. The LX8380 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 LX8380 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 LX8380 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 23 summarizes the local memories that can be integrated with the LX8380.

Table 23: 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.

6.2. 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 LX8380. When the LX8380 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 6.5. 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 LX8380 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.

6.3. 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.

6.4. Instruction Cache (ICACHE) LMI

The ICACHE LMI supplies the interface for a direct mapped or two-way set associative instruction cache attached to the LX8380 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 24.

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 24: 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 25 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 24.

Table 25: 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.

6.5. 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 6.2, 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 26.

Table 26: 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 27 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 26.

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 LX8380 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 LX8380. The size of the memory region must be a power of two, and must be naturally aligned.

Table 27: 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.

6.6. Data Cache (DCACHE) LMI

The DCACHE LMI supplies the interface for a data cache attached to the LX8380 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 28.

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 28: 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 29 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 28.

Table 29: 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 30 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 30: 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

6.7. Scratch Pad Data Memory (DMEM) LMI

The DMEM LMI supplies the interface for a scratch pad data RAM attached to the LX8380 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.

LX8380 applications may optionally specify the use of a 128-bit data memory width through an *lconfig* setting. When RAM BIST or scan collars are enabled with *lconfig*, LX8380 does *not* tie the DMEM RAM into the RAM BIST paths or scan collar muxes. Other RAMs remain connected to these options.

The DMEM configurations and the synchronous RAMs required for each are summarized in the Table 31.

For LX8000, dual-port RAM may optionally be used for DMEM. The second port is brought out the processor hierarchy for customer connection.

Table 31: 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	dual port 128 x 64 bits	dual port 64 x 128 bits
2K bytes	dual port 256 x 64 bits	dual port 128 x 128 bits
4K bytes	dual port 512 x 64 bits	dual port 256 x 128 bits
8K bytes	dual port 1,024 x 64 bits	dual port 512 x 128 bits
16K bytes	dual port 2,048 x 64 bits	dual port 1,024 x 128 bits
32K bytes	dual port 4,096 x 64 bits	dual port 2,048 x 128 bits
64K bytes	dual port 8,192 x 64 bits	dual port 4,096 x 128 bits
128K bytes	dual port 16,384 x 64 bits	dual port 8,192 x 128 bits
256K bytes	dual port 32,768 x 64 bits	dual port 16,384 x 128 bits

Table 32 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 31.

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 LX8380. The size of the memory region must be a power of two, and must be naturally aligned.

Table 32: DMEM RAM Interfaces

Signal	Description
DW_DATAINDEX	Decoded data RAM index.
DWR_DATARD	Data RAM read data.
DW_DATAWR	Data RAM write data.
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).
DMADW_RCLK	Data RAM dual port DMA clock (optional).
DMADW_DATAINDEX	Decoded data RAM index.
DMADW_DATARD	Data RAM dual port DMA read data.
DMADW_DATAWR	Data RAM dual port DMA write data.
DMADW_DATAWE<N>	Data RAM dual port DMA write enable.
DMADW_DATARE<N>	Data RAM dual port DMA read enable.
DMADW_DATACS<N>	Data RAM dual port DMA chip select.

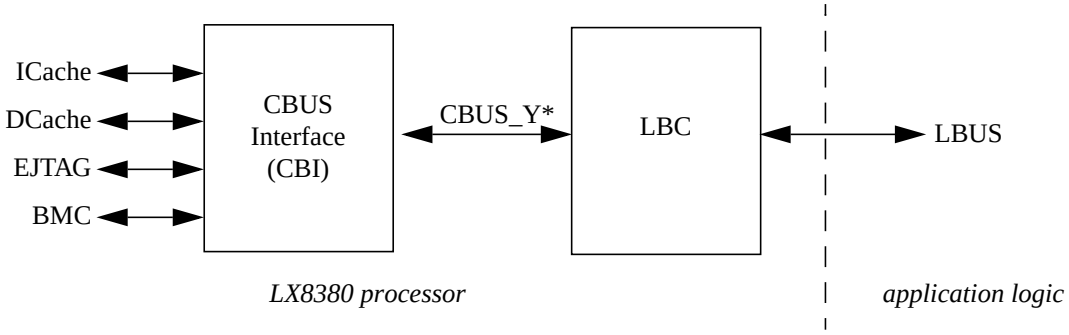
Note: <N> designates an available active-low version of a signal.

7. CBUS Interface

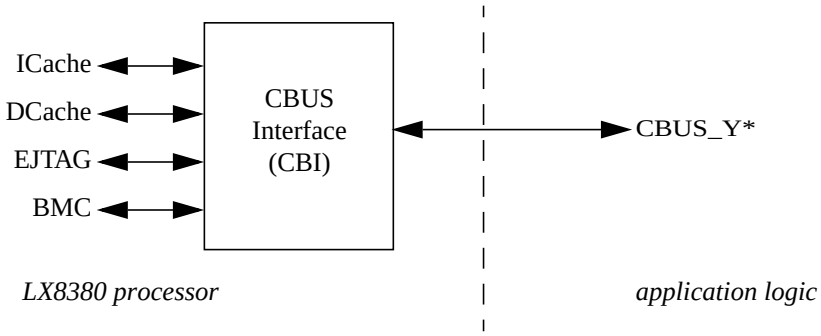
This section describes the CBUS, a system interface to the LX8380 that is an alternative to the LBUS. The CBUS Interface (CBI) provides a simple signalling layer between the LX8380 processor's cache controllers and the optional LX8380 system bus interface, the LBC. (See Section 8 for information on the LBC and LBUS.) LX8380 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 LX8380 using CBUS.

7.1. System Interface Configuration

Figure 10 illustrates the LX8380's organization for the different system interface configurations. This section describes the configuration shown is part (b) of the figure.



(a) LX8380 Configured with LBUS Interface



(b) LX8380 Configured with CBUS Interface

Figure 10: LX8380 System Interface Configurations

7.2. 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 and split read 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. LX8380 applications that employ LBUS instead of CBUS still use the CBI write buffer.

When the CPU issues a (non-split) 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 or BMC 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 or BMC 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.

7.3. 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 LX8380 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 LX8380 is configurable to work with either mode.

The LX8380 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 33. 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 LX8380 is configurable to work with 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

7.4. CBUS Byte Alignment

CBUS data must be driven to the byte lanes according to the rules shown in Table 34. Alignments not shown are not generated by the CPU. All multi-beat operations transfer multiple twin word beats over CBUS.

Table 34: 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

7.5. CBUS Interface Signal List

Table 35 summarizes the LX8380's CBUS signals.

Table 35: 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_YSPLTO	O	1=Split 0=normal transaction
CBUS_YLTIDO[3:0]	O	Local thread ID
CBUS_YUCO	O	1 - uncached access, 0 - cached access
CBUS_YSRCO[3:0]	O	transaction source (within LX8380): 4'b0001 Instruction Cache 4'b0010 Data Cache or EJTAG DMA write 4'b0100 EJTAG DMA read 4'b1000 BMC
CBUS_YDBUSYO	O	1 - LX8380 is not ready to receive Data. Any return with CBUS_YVALTYPEI of Data (4'b0010) is ignored by the LX8380. External logic must hold such data until CBUS_YDBUSYO is deasserted. 0 - LX8380 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_YLTIDI[3:0]	I	Thread associated with Read Data

Name	I/O	Function
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 BMC
CBUS_YSPLTSZI[2:0]	I	Size of split Read Data beat: 3'b000 - 1 byte 3'b001 - 2 bytes 3'b011 - 1 word 3'b100 - 2 words
CBUS_YIDLEI	I	Indicates external CBUS_Y device is in an idle state, i.e. has no pending read or write transactions.

7.6. CBUS Transaction Types

The following transaction types are supported by the CBUS interface:

1. Single split/normal read.
2. Line split/normal read.
3. Single write with split read.
4. Single write.
5. Line writes.

7.7. 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 LX8380 to initiate an access to external logic. Additional CBUS_Y* outputs are driven by the LX8380 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, BMC or EJTAG DMA data is present on CBUS_YDATAI, the data is always accepted by the LX8380.
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.

7.8. 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.

7.8.1. 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 LX8380 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.

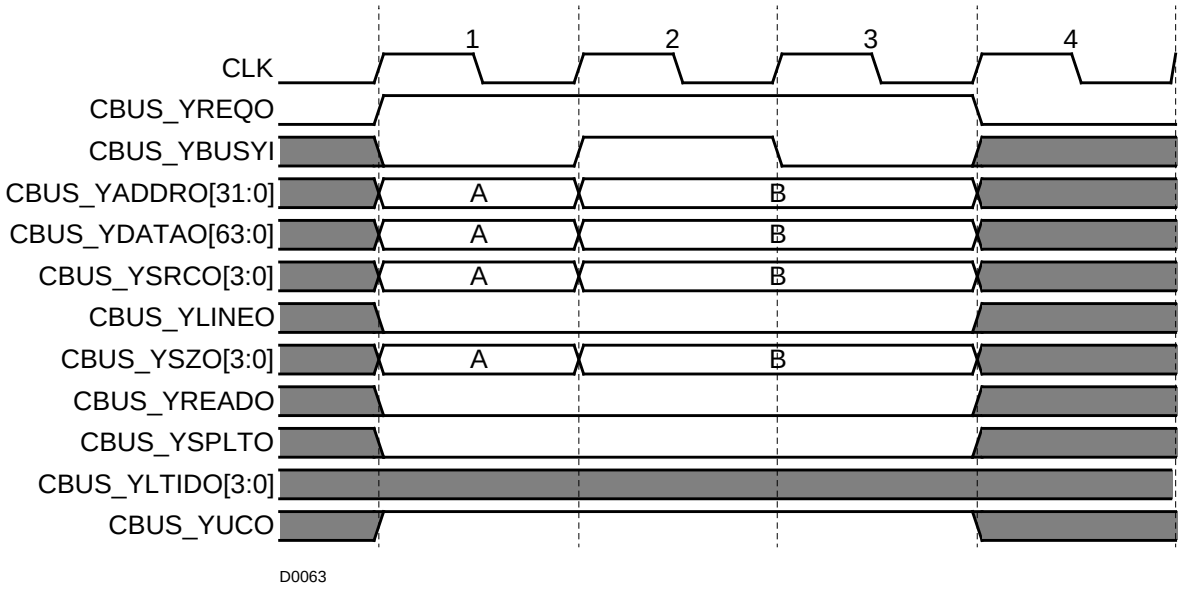


Figure 11: CBUS Back-to-Back Single Writes with Busy

7.8.2. 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.

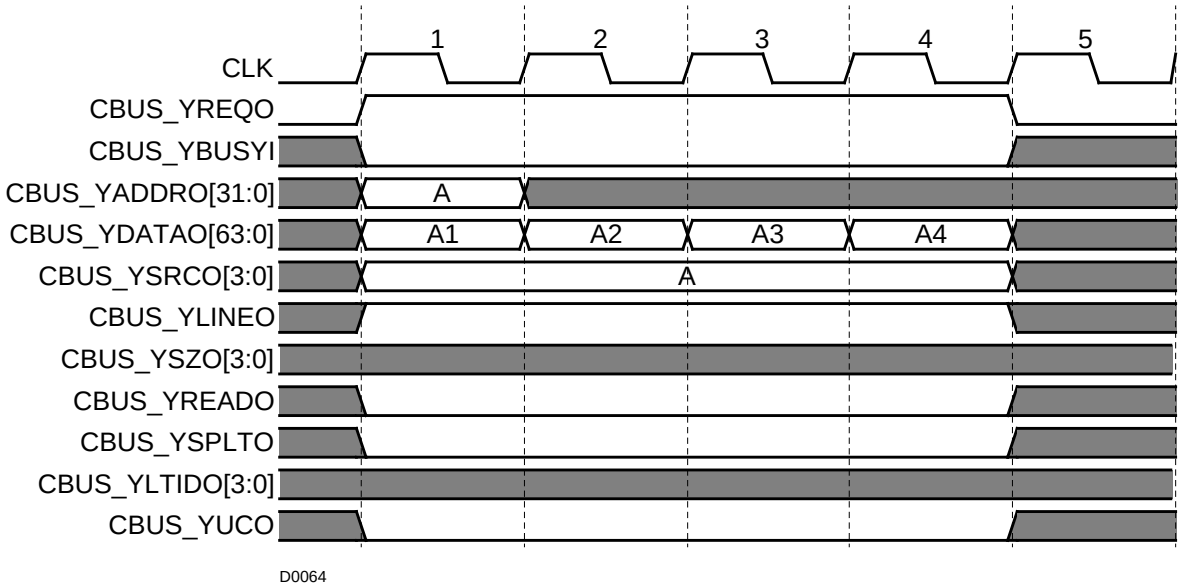


Figure 12: CBUS Line Write

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

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

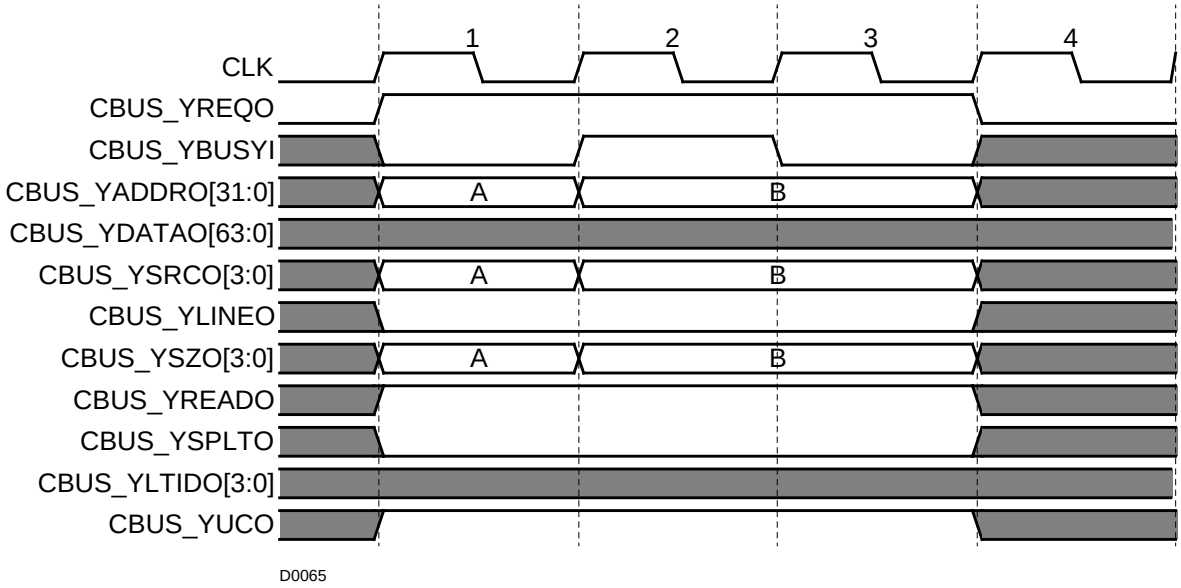


Figure 13: CBUS Back-to-Back Single Read Requests with Busy

7.8.4. Line Read Request

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

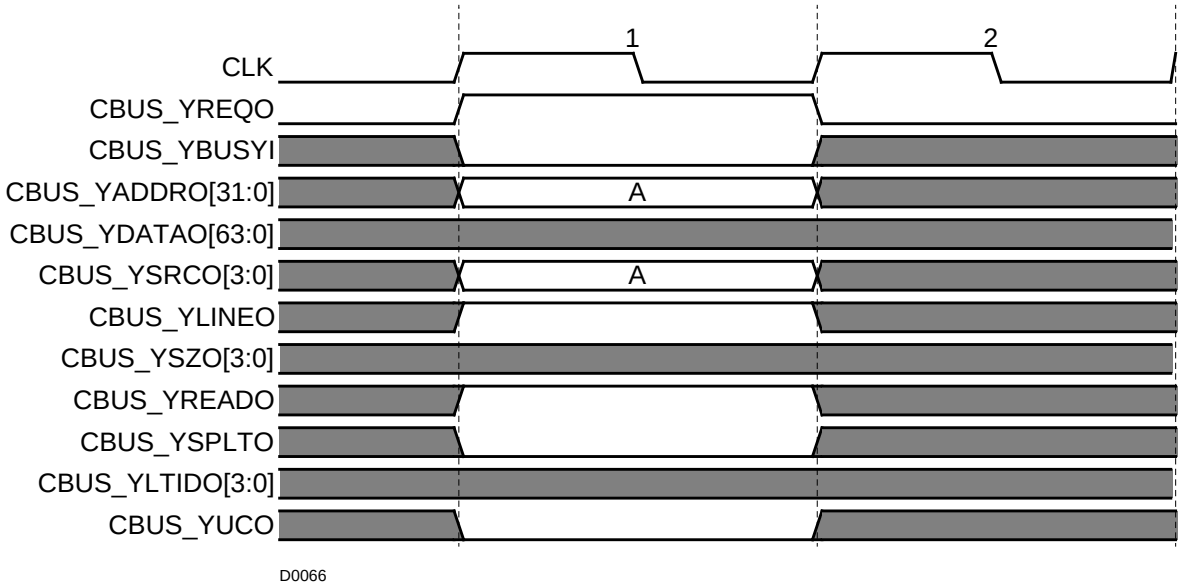


Figure 14: CBUS Line Read Request

7.8.5. Split Read Request

The LX8380 always issues a split data read request when an LW.CSW or LT.CSW or LQ.CSW instruction is executed, and may also be configured via *lconfig* to issue split reads for all read operations. The LX8380 may initiate a new transaction in the following cycle. The processor may have multiple split transaction reads outstanding, as defined in Table 36.

Table 36: Maximum Number of Outstanding Split Reads

Split Read Source	CBUS_YSRCO[3:0]	Maximum Reads Outstanding
Instruction	4'b0001	One
Data read	4'b0010	One per processor context
EJTAG DMA read	4'b0100	One
BMC read	4'b1000	Limited only by the assertion of the CBUS_YBUSYI input.

To initiate a split read transaction, the processor asserts CBUS_YSPLTO, CBUS_YREQO and CBUS_YREADO. This can occur for word, twinword, quadword and line reads. The CBUS_YLTIDO[3:0] output indicates the processor context number or BMC channel number associated with the request. The device that services the split read request is responsible for retaining the context number while it performs the read operation, and re-sourcing the context number when the read data is finally returned.

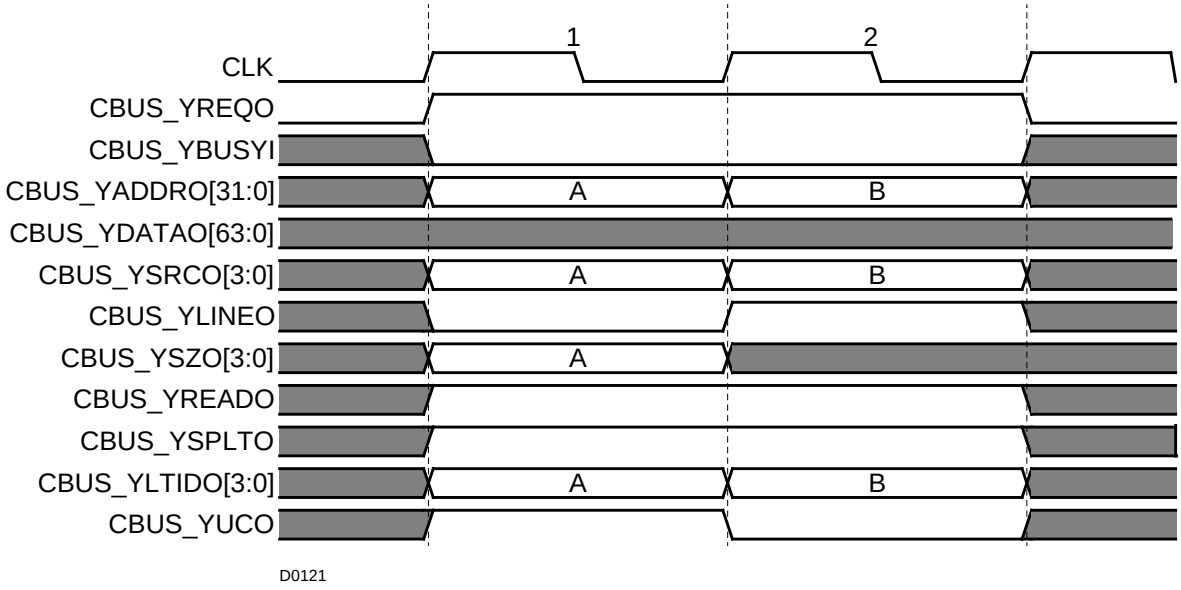


Figure 15: CBUS Split Read Requests

7.8.6. Write with Split Read Request

The processor issues a write with split data read request when a Write Descriptor with Split Load instruction (WDLW.CSW, WDLT.CSW or WDLQ.CSW) is executed. This transaction supplies the write data and initiates the read request in a single cycle. The processor may initiate a new transaction in the following cycle. The limitations on outstanding split Data reads, defined in Table 36, apply to the outstanding read transactions that are generated in this way.

To initiate a write with split read transaction, the processor asserts CBUS_YSPLTO and CBUS_YREQO, and de-asserts CBUS_YREADO. This can only occur for word, twinword and quadword reads. The CBUS_YSZO[3:0] output indicates the size of the read request. The write data size is always a twinword. The CBUS_YLTIDO[3:0] output indicates the processor context number or BMC channel number associated with the request. The device that services the split read request is responsible for retaining the context number while it performs the read operation, and re-sourcing the context number when the read data is finally returned.

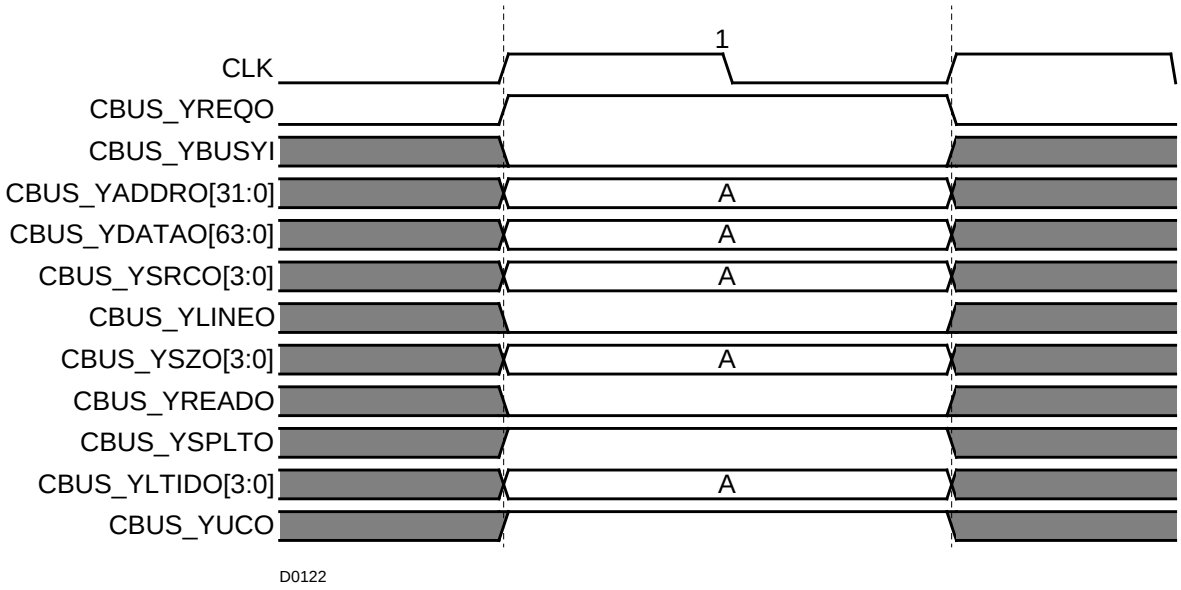


Figure 16: CBUS Write with Split Read Request

7.8.7. Returning Read Data

External logic supplies read data on the CBUS_YDATAI and CBUS_YLTIDI inputs while asserting a bit within CBUS_YVALTYPEI. If CBUS_YVALTYPEI indicates Data (4'b0010), the LX8380 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

If the read data is in response to a split read request, the external logic must also drive CBUS_YSPLTSZI to indicate the size of the data beat.

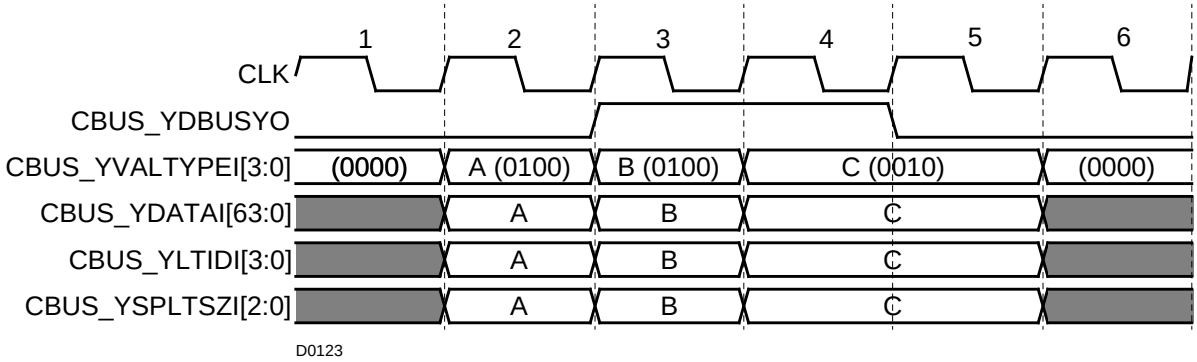


Figure 17: CBUS Read Data and DBUSY

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.

If the line read data is in response to a split read request then each data beat is a twinword, and the external logic must therefore drive CBUS_YSPLTSZI to 3'b100.

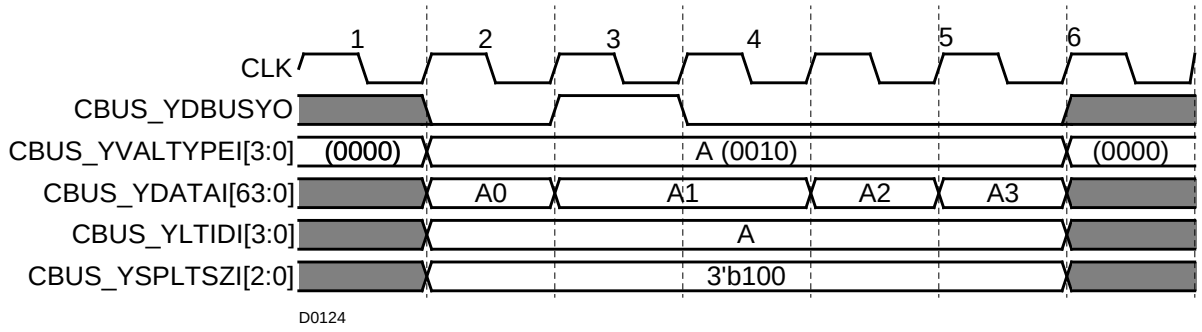


Figure 18: Read Data for a Line Read Request

7.8.8. Latency of CBUS Transactions

Figure 19 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. (But note, the CPU is typically not stalled during split read transactions that are issued as a result of the L*.CSW or WDL*.CSW instructions.)

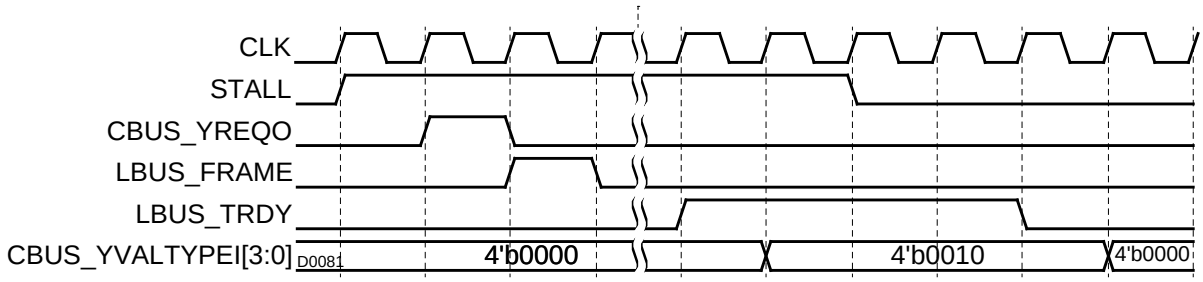


Figure 19: Latency of CBUS Transactions.

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.

8. Lexra System Bus (LBUS)

This section describes the optional Lexra System Bus (LBUS) and the Lexra Bus Controller (LBC) that connects the LX8380 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 LX8380's CBUS interface rather than the LBUS. (See Section 7, CBUS Interface.)

8.1. Connecting the LX8380 to Internal Devices

The Lexra Bus Controller (LBC) provides the connection between the LX8380 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.

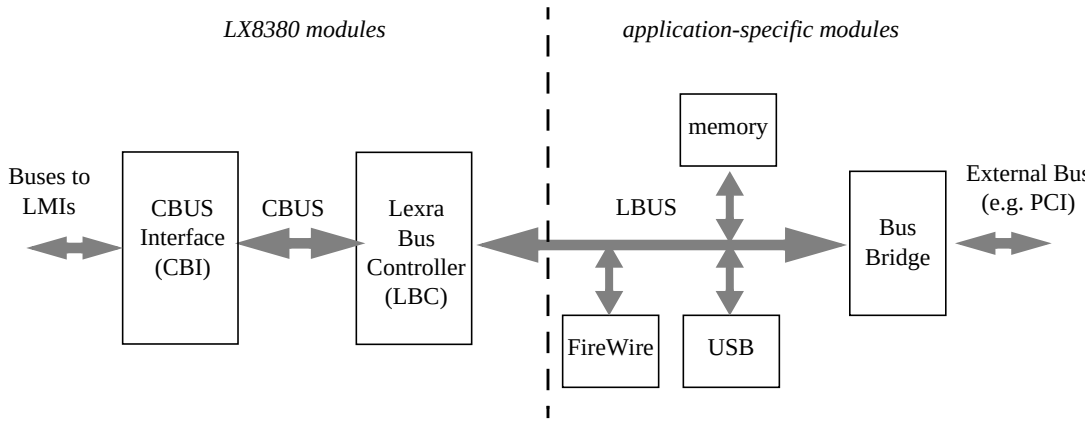


Figure 20: Lexra System Bus (LBUS) Diagram

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 LX8380 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.

8.2. 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.

8.3. Bus Operations

The purpose of the Lexra bus is to connect together the various components of the system, including the LX8380 CPU, main system memory, I/O devices, and external bus bridges. Different devices have different transfer requirements. For example, the LX8380 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
Split Read
Write Split Read
Split Data

8.3.1. 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.)

8.3.2. 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 LX8380 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 LX8380 is configurable to work with either mode of operation.

The LX8380 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 8.5. The LX8380 is configurable via *lconfig* to generate bus commands for either mode.

Table 37: 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

8.3.3. 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 LX8380 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.

8.3.4. Single Data Write

The single data write operation writes a twinword, a single word, a halfword, or a byte to the target.

The LX8380 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.

8.3.5. 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 LX8380 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.

8.3.6. 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 LX8380.

8.3.7. Split Read

The LBC issues a Split Read command when the processor executes an LW.CSW, LT.CSW or LQ.CSW instructions. If the LX8380 is configured for split line reads (via *lconfig*), the processor issues a Split Read command for all data and instruction read transactions, including line reads.

The Split Read bus transaction terminates when the target has accepted the command by asserting TRDY. The bus is free for other operations while the target device performs the read internally. When the target is ready to supply the read data, it issues a Split Data or Split Line Data command as a bus master, described below.

8.3.8. Write Split Read

The LBC can issue a Write Split Read command when the processor executes a WDLW.CSW, WDLT.CSW or WDLQ.CSW instruction. This bus command writes 64-bit data to a device while simultaneously making a split read request. CMD[3:0] specify the size of the read request, one, two or four words. The Write Split Read bus transaction terminates when the target device has accepted the command and the write data by asserting TRDY. The bus is free for other operations while the device performs the write and read operations internally. When the target device is ready to supply the read data, it issues a Split Data command as a bus master, described below.

8.3.9. Split Data

A target device that has accepted a Split Read or Write Split Read command supplies the data to the requestor using the Split Data command. The device saves the GTID obtained from the Split Read or Write Split Read command, and performs the read operation internally. When the device is ready to supply the read data, the device acts as an LBUS master device. It issues a Split Data command that identifies the original requestor's GTID, and supplies the read data with the command. The LBC that matches the GTID will act as LBUS target device and accept the data. An LBC acts as a target only for Split Data commands.

8.4. Signal Descriptions

Table 38: LBUS Signal Description

Signal Name	Source (Initiator/Target/Ctrl)	Description
BCLOCK	Ctrl	Bus Clock
BCMD[8:0]	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).
BGTID[15:0]	Initiator	For all transactions except Split Data, indicates the Global Thread ID of the initiator. For Split Data transactions, indicates the Global Thread ID of the target to which the data is directed.

8.5. LBUS Commands

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

BCMD[8:6]	000	read
	001	write
	010	split read
	011	write split read
	100	reserved
	101	split data
	110	reserved
	111	reserved
BCMD[6]	0	read
	1	write
BCMD[5:4][a]	00	burst, fixed length[b]
	01	burst, unlimited number of words
	10	line, interleaved wrap[c]
	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. If the processor is not configured to issue Split Read commands for all read operations, CMD[5:4] is always 00 for Split Read, Write Split Read and Split Data commands.
b. For burst transfers, the length is determined BCMD[3:0]
c. For line transfers the length is determined by the RTL line size configuration (set with *lconfig)*, not BCMD[3:0]

8.6. LBUS Byte Alignment

LBUS data must be driven to the byte lanes according to the rules shown in Table 39. Alignments not shown are not legal. All multi-beat operations transfer multiple twin word beats over LBUS.

Table 39: 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.

8.7. Split Transactions

The processor generates a Split Read command as a result of executing specific instructions such as LW.CSW. These instructions also cause the processor to perform a context switch. The split transaction makes efficient use of the system bus while the processor executes instructions for a different context. The LX8380 can also be configured with *lconfig* to issue Split Read commands for all read operations, including line reads that are initiated as a result of an instruction cache or data cache miss. Although the processor does not perform a context switch in these cases, the split transaction can still improve system level performance by making more efficient use of the system bus.

The split transactions are divided into two parts. The first half, initiated by an LBC, requests data from a target device (either with a Split Read or Write Split Read command). Unlike a regular read request, the LBC does not hold the bus until the read data is returned. Once the LBC knows that the target has received the split read request, it releases the bus and waits for the data to be returned at a later time. The data is returned to the LBC with a Split Data command from the LBUS device that accepted the split read request.

When the processor executes a LW.CSW, LT.CSW or LQ.CSW instruction, a Split Read command for one,

two or four words will be issued on the LBUS. CMD[3:0] is used to indicate the size of the read data requested. No data will be transferred with the request. When the processor executes a WDLW.CSW, WDLT.CSW or WDLQ.CSW instruction, one LBUS transaction (Write Split Read command) will issue the write data and a split read request. Unlike a regular write, the size of the write data is always two words. CMD[3:0] indicates the size of the requested read data.

If the processor is configured to issue Split Read commands for all read operations, an instruction or data cache miss will cause the processor to issue a Split Read command that specifies a line transfer, via CMD[5:4]. The line size is configuration dependent, and is not conveyed in CMD[3:0].

Once an LBUS target device has accepted the split read request, it must return data at a later time. The target acts as an LBUS master device and initiates a Split Data command. The target LBC accepts this command and receives the data.

The Global Thread ID (GTID) is driven with each split read request. The target device retains the GTID value associated with the split read request, and supplies this value when it initiates the split data transaction to return the data. This allows an LBC to identify split read data that is targeted for it, and allows the processor associate the data with the correct context. A target system bus device must save all 16 bits of the GTID value driven with a split read request, including Reserved fields, and subsequently drive the 16-bit value onto GTID during the split read data transaction.

Table 40: LBUS GTID Fields

GTID[15]	GTID[14:13]	GTID[12]	GTID[11:4]	GTID[3:0]
Inst	Reserved	BMC	ProcNum	ContextNum

Field Name	Interpretation
Inst	0 - Data related request.. 1 - Instruction releated request.
Reserved	Implementation specific.
BMC	0 - ContextNum identifies a CPU context number. 1 - ContextNum identifies a BMC channel number.
ProcNum	Procesor Number
ContextNum	Within-processor context number or BMC channel number.

The LBC drives GTID all transactions, including non-split transaction types.

8.8. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the element of the LX8380 that connects to the Lexra Bus. It forwards all transaction requests from the LX8380 CPU to the Lexra Bus.

8.8.1. LBC Commands

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

Table 41: LBUS Commands Issued by the LBC

Command	BCMD[8:6]	BCMD[5:4]	BCMD[3:0]	Circumstances
Read Line (non-split)	000	10 or 11, depending on configuration	undefined	A cache miss during a read by the CPU, and the CPU is not configured to use split transactions for all reads.
Read Single (twinword/word/halfword/byte)	000	00	10xx or 1100	A read by the CPU from an address in uncachable address space, and the CPU is not configures to use split transactions for all reads.
Write Line	001	10 or 11, depending on configuration	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)	001	00	10xx	A write by the CPU into cacheable or uncachable address space.
Split Read	010	00 for Split Read Single, 10 or 11 for Split Read Line.	1011, 1100 or 0000	An LW.CSW, LT.CSW or LQ.CSW instruction is executed. Also performed for all bus read operation if the CPU is configured to use split transactions for all reads.
Write Split Read	011	00	1011,1100 or 0000	A WDLW.CSW, WDLT.CSW or WDLQ.CSW instruction is executed.

8.8.2. Write Buffer

The LX8380 includes a write buffer in its CBUS Interface. When the LX8380 is configured to include the LBC, the CBUS Interface and its write buffer are always included as an internal LX8380 module. See Section 7.2, for a description of the write buffer.

8.8.3. 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 non-split read data as fast as the LBC can transfer it. There is no need for a large read buffer if split read transactions are not employed. 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.

8.9. 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 8.12 on page 107).

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

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

signal. Either agent may de-assert its ready signal to indicate that it cannot source or accept data on this particular clock edge.

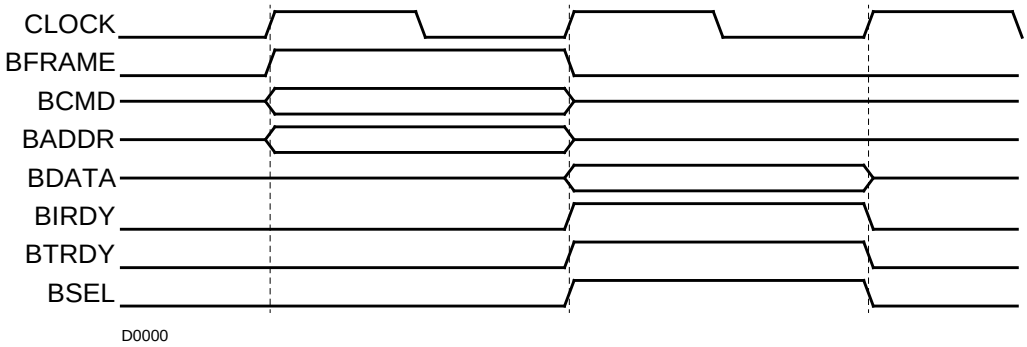
6. When the initiator is ready to send or receive the last data beat, that is, when it asserts BIRDY for the last time, it also de-asserts BFRAME. It will de-assert BIRDY when the last data beat is transferred.

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.

8.9.1. 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. The LBC does not issue non-split Line Read transactions if the processor is configured to employ split reads for all read operations. Instead, a Split Read is issued.



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.

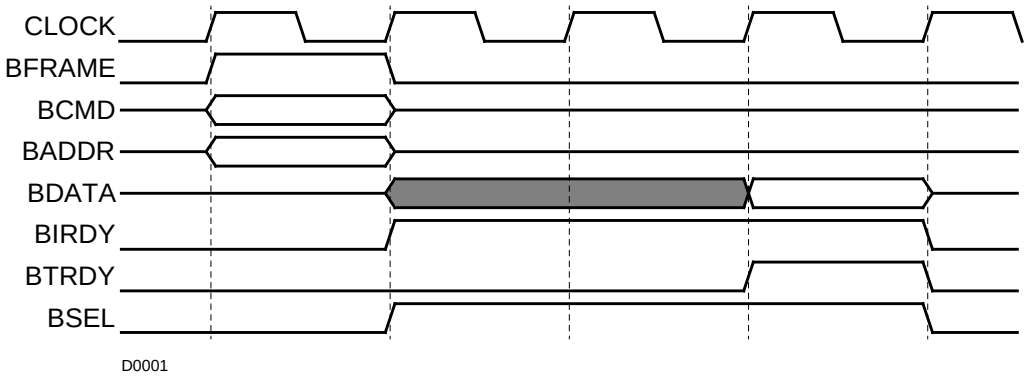
1. Initiator asserts BFRAME and drives BADDR.

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.

8.9.2. 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.

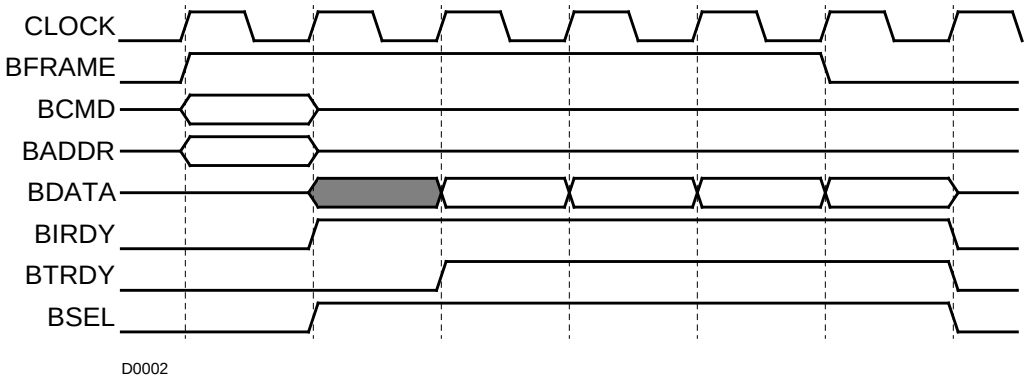


This is a common single data read operation.

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.

8.9.3. 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. The LBC does not issue non-split Line Read transactions if the processor is configured to employ split reads for all read operations. Instead, a Split Read is issued.



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.

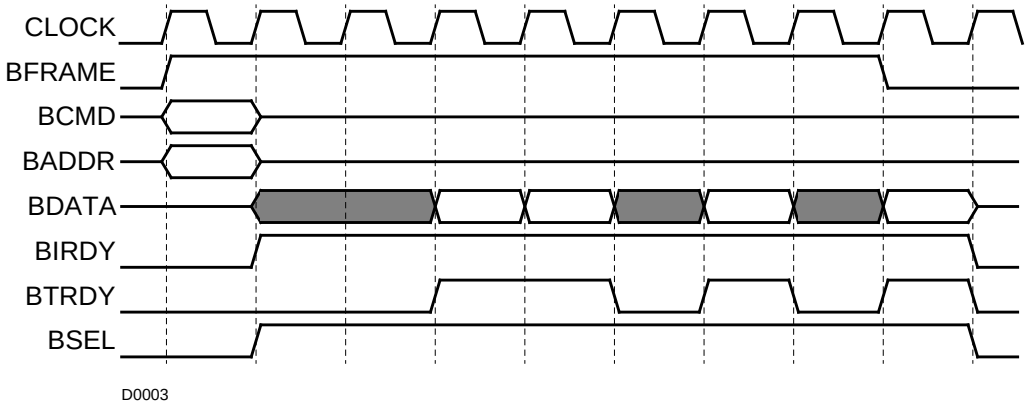
5. Target drives third data beat and continues to assert BTRDY.

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.

8.9.4. 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.

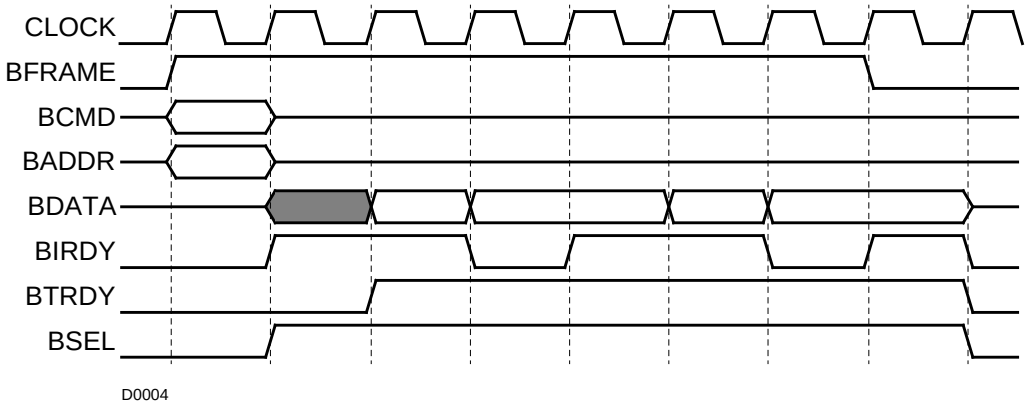
7. Target drives third data beat and asserts BTRDY.

8. Target cannot get fourth data beat, so it de-asserts BTRDY.

9. Target drives fourth data beat and asserts BTRDY.

8.9.5. 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.

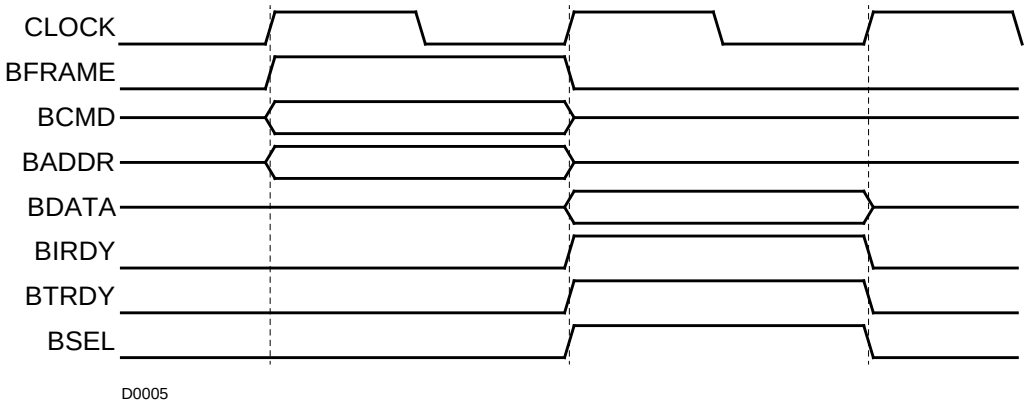
8.9.6. Burst Read

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.

8.9.7. Single Data Write with No Waits

A single data write operation occurs when the LX8380 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 LX8380 does not generate twinword writes.)

A single data write without waits requires two cycles.



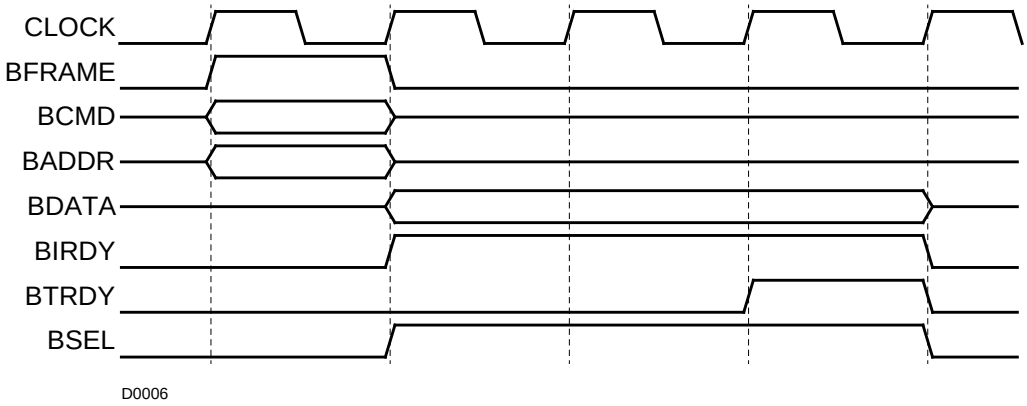
1. Initiator asserts BFRAME and drives address.

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.

8.9.8. 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.

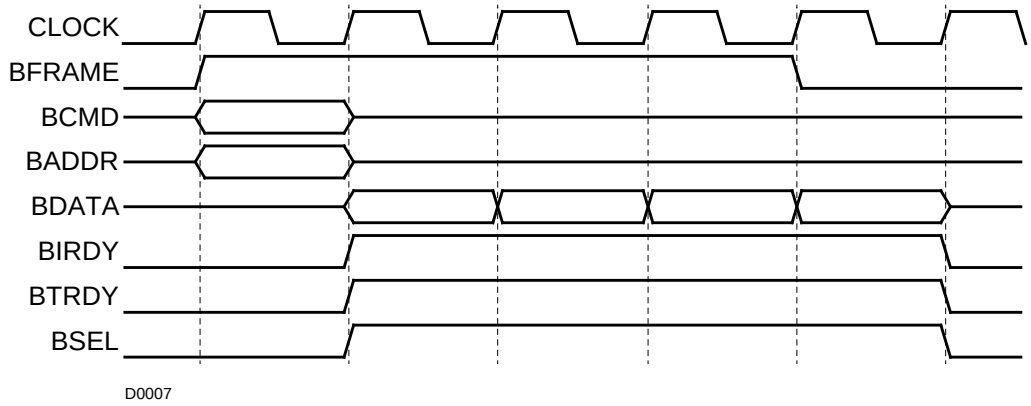


This is the same description as the above example, except that the target inserts two wait states until it asserts BIRDY to indicate acceptance of data.

8.9.9. 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

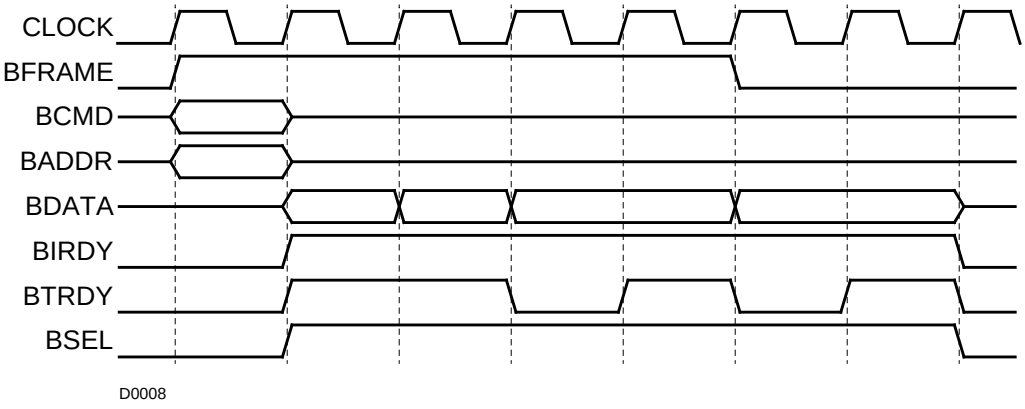
following illustrates a best-case scenario with no wait states.



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.

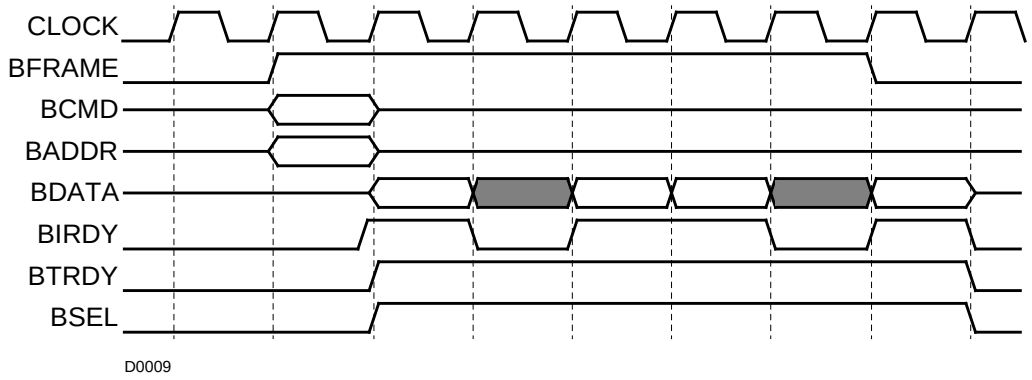
8.9.10. Line Write with Target Waits

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.



8.9.11. Line Write with Initiator Waits

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

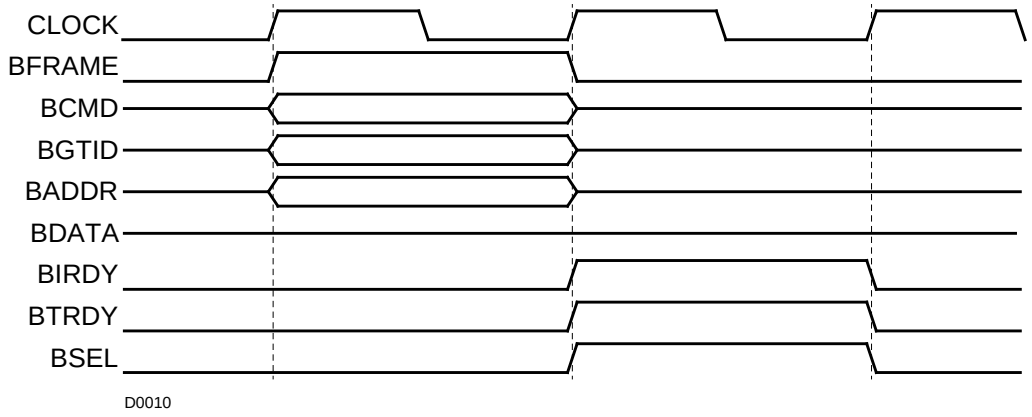


8.9.12. Burst Write

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.

8.9.13. Split Read command

The processor may be configured with *lconfig* to issue Split Read commands for all read operations. Otherwise, the LBC issues a Split Read command only when the processor executes an LW.CSW, LT.CSW or LQ.CSW instruction. The following is an example of a single word read request.



1. An LBC initiates the transaction by asserting FRAME and driving the ADDR for the transaction. It drives the CMD bus with the Split Read command. GTID is also driven by the LBC with the Processor/Context Number information.

2. The target decodes the address and asserts SEL and TRDY to respond to the request. TRDY should always be asserted with SEL. It saves the ADDR and GTID information which it will use when it returns the data. No data needs to be transferred, so the data bus is inactive. FRAME is de-asserted and ADDR, CMD, GTID are not driven. IRDY is asserted.

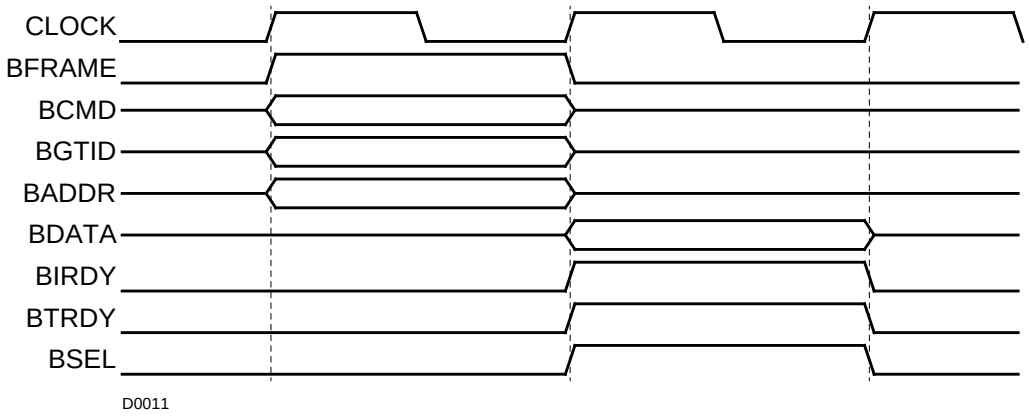
3. The LBC de-asserts IRDY and the target device de-asserts SEL and TRDY to indicate the split read request transaction is complete.

There are no data stalls allowed since no data is being transferred. The target should assert TRDY as soon as it asserts SEL.

The first half of the read transaction is now complete. The LBC will wait for the target device to return the requested data using the Split Data command.

8.9.14. Write Split Read

When the processor executes a WDLW.CSW, WDLT.CSW or WDLQ.CSW instruction, the LBC issues a write command with split read request. With this command, the LBC writes data to a device while simultaneously making a split read request. The write data consists of two words. The requested read data size may be 1, 2 or 4 words, indicated by CMD[3:0].



1. An LBC initiates the transaction by asserting FRAME and driving the ADDR for the transaction. It drives the CMD bus with a Write Split Read command and CMD[3:0] indicates either one word or two word split read request. GTID is driven with the Processor/Context Number information.

2. The target decodes the address and asserts SEL. In this example the target is immediately ready to accept the write data so it also asserts TRDY. It saves the GTID information which it will use when it returns the data. The LBC de-asserts FRAME since this is a single cycle write. It also drives IRDY and the DATA bus. ADDR, CMD and GTID are only driven the first cycle.

3. The LBC de-asserts IRDY and the target device de-asserts SEL and TRDY to indicate the write transaction has completed. The read request has also been transferred and the target must issue a data response at a later time.

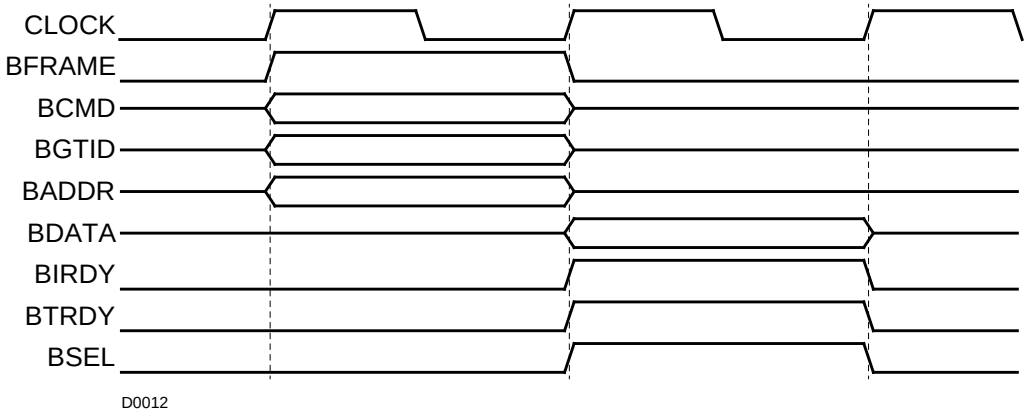
The transaction will look the same for a split read request of two words, except CMD[3:0] will indicate a two word request instead of one word.

Since write data is being transferred with these transactions, data stalls are allowed. The rules for TRDY and IRDY are the same for these write transactions as they are for regular write transactions.

When an LBC issues a Split Read or Write Split Read command and successfully completes the request to the target, the LBC will consider that operation complete. It is the responsibility of the target device to return data to the LBC by issuing a Split Data command.

8.9.15. Split Data

Once an LBC has sent a split read request (either with a Split Read or Write Split Read command) and the target device has accepted the request, the device must supply the requested data and return it to the LBC. To do this, it must act as an LBUS master device and initiate a Split Data command. The LBC which originated the Split Read will act as LBUS target device and accept the data. An LBC only acts as a target for Split Data commands.



1. The LBUS device that accepted the read request now asserts FRAME to indicate it is ready to return the requested data. It drives CMD with the Split Data command. CMD also indicates the transaction size. The GTID bus is driven with the correct Processor/Context information. The ADDR bus must be driven with the address of the requested read data.

2. Each LBC examines the GTID bus to determine which Processor this data is for. The LBC that is associated with the ProcNum asserts SEL and TRDY to accept the two words of data. FRAME is de-asserted while IRDY is asserted. ADDR, CMD and GTID are only driven the first cycle.

3. The master device de-asserts IRDY and the LBC de-asserts SEL and TRDY to indicate the transaction is complete.

The LBC then returns the read data to the context that requested it.

The CMD encoding indicates the transaction size. If the processor is configured to issue Split Read commands for all read operations, the split read data size may be 1, 2, 4, 8 or 16 bytes, or a cache line. Otherwise the split read data size is limited to 4, 8 or 16 bytes. The data is aligned on the data bus based on the original read address, according to the rules shown in Table 39 on page 93.

Data stalls are allowed during data response transactions. An LBC will properly handle data stalls on the bus, and may de-assert TRDY to stall the transaction itself. For performance reasons, the Read Buffer in the LBC should be large enough to avoid this.

8.10. Ordering Rules with Split Transactions

The LBC follows the same rules for allowing a Split Read request to be issued as it would a standard read request.

Once an LBC has issued the Split Read or Write Split Read command, it does not keep track of the read request. This means a subsequent write transaction could be issued to the same address before the requested data has been returned to the LBC. The LBC will not stall the write or try to enforce any coherency in this case.

If more than one Split Read/Write Split Read request is outstanding on the LBUS, the corresponding data responses do not have any ordering requirements. The LBC will use the GTID that was presented with the split data to return the data to the correct context.

8.11. LBC Signals

The table below summarizes the LX8380 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 42: 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
output	LSelO	BSEL	LBC slave select
input	LSel	BSEL	System slave select
output	LTrdyO	BTRDY	LBC target ready
input	LTrdy	BTRDY	System target ready
output	LCmd[8:0]	BCMD[8:0]	LBC command
input	LCmdI[8:0]	BCMD[8:0]	System command
output	LGTidO[15:0]	BGTID[15:0]	LBC global thread ID
input	LGTidI[15:0]	BGTID[15:0]	System global thread ID
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

8.12. Arbitration

8.12.1. 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.

8.12.2. 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.

8.13. 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 LTRdyO, LSelO, LFrame and LIrdy.

LCoe qualifies LCmd, LAddrO and LGTidO.

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.

9. Block Move Controller (BMC)

9.1. BMC Overview

The BMC performs arbitrary length transfers between DMEM and system devices such as main memory. The transfer length may be 1-262,144 (256K) bytes, with byte granularity. The BMC breaks the transfer into a series of transaction requests which are presented to the Data Local Memory Interface (DLMI). The DLMI uses its CBUS interface to access the system device.

Each BMC transfer is assigned to a *channel*. The BMC supports up to 16 channels. A hardware register set is provided to setup and control each channel's transfer. In the LX8380, each channel is usually associated with a processor context.

BMC registers are accessed via the coprocessor 3 interface. Software requests a BMC transfer by performing move operations to the register set. Register contents persist between transfer requests, reducing the number of instructions required to request subsequent transfers. Software typically performs a context switch after making a transfer request.

The BMC informs the CPU when a transfer completes either through a flag that can be polled by software, an interrupt, or by clearing the appropriate CPU Wait-Event bit to activate an idle context.

The DLMI provides the data interconnect between CBUS, DMEM, DCACHE, and the CPU. Requests for data transfers may be initiated by the CPU or the BMC. These requests are processed by the DLMI and passed to the CBUS. The DLMI can be configured to provide either a one or two-port DMEM interface. With a one port interface, CPU and BMC transfers share the port, and an arbitration scheme is used. With a two port interface, the CPU and BMC each have a dedicated port.

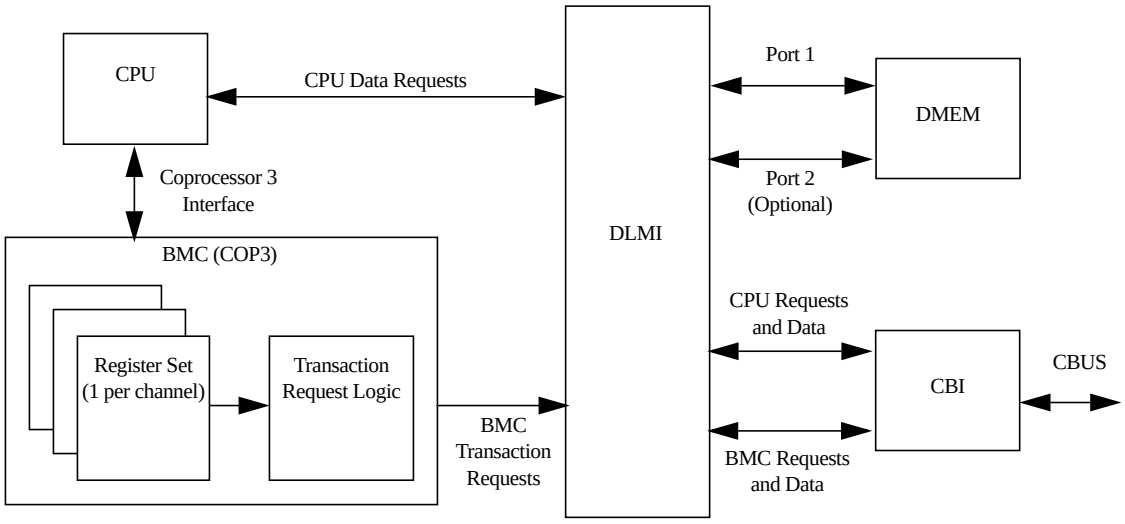


Figure 21: Block Move Controller

9.2. Transfers

The BMC is controlled via the coprocessor 3 interface. The BMC has one set of registers for each channel. Software requests a BMC transfer by performing move operations to a channel's register set. Register contents persist between transfer requests, reducing the number of instructions required to request subsequent transfers.

Software may request read or write transfers. Write transfers move data from DMEM to a device. Read transfers move data from a device to DMEM. Software typically performs a context switch after making a transfer request via the CSW instruction.

Software must not change the values written to a channel's register set while a transfer request from that channel is pending. Software may update a channel's register set after notification of transfer completion.

Transfer requests are assigned to a transaction *class* via a field in each channel's register set. Within a single transfer class, transfers requests are processed sequentially, with the oldest request processed first. Across transfer classes, transfers are interleaved. Transfers may be interleaved because they are broken into *transactions*, described in the next section. Interleaved transfers are processed in a round-robin fashion.

Interleaving transfers generally improves system performance, assuming that the interleaved transfers target different devices. Since any one device has more time to process a transaction, it is less likely to stall the system bus. Interleaved transfers may not be appropriate if the transfers target the same device. For example, in a streaming device, transactions associated with a transfer must occur in sequence. In this case, all transfers targeting the streaming device must share the same class.

9.3. Transactions

The BMC breaks each transfer into a series of transaction requests. The transaction size may be byte, word, double word, or line. There are separate control bits to disable word, double word, and line transactions.

When a read transfer is requested, the resulting transactions are split reads. The DLMI allows multiple outstanding split read transactions in order to fully utilize the system bus.

The size of a transaction is selected based on the CBUS transfer address and transfer length. The largest enabled transaction size that meets the following criteria is selected:

- The transaction size must be aligned to the current CBUS transfer address
- The transaction size must not cause the requested transfer length to be exceeded

Here is an example of this process:

- Transfer size: 91 bytes
- word transactions disabled
- CBUS transfer starting address: 0x0000_000f
- Line size: 32 bytes

Transaction Address	Transaction Size
0x0000_000f	byte
0x0000_0010	double word
0x0000_0018	double word
0x0000_0020	line
0x0000_0040	line
0x0000_0060	double word
0x0000_0068	byte
0x0000_0069	byte

An option is provided to maintain a constant CBUS address. In this case, the transaction size is fixed at the largest enabled transaction size. If the transfer size requested is not an integer multiple of this transaction size, the final transaction is padded. The padding bits consist of whatever follows the transfer data in DMEM.

Here is an example showing the transaction requests generated:

- Line transactions disabled
- CBUS transfer starting address: 0x0000_0008
- Transfer size: 26 bytes

Transaction Address	Transaction Size
0x0000_0008	double word
0x0000_0008	double word
0x0000_0008	double word
0x0000_0008	double word

9.4. Transaction Sequence Due to Transfer Class

The examples in the previous section show sequences of sequential transaction requests that are all associated with the same transfer. Transaction requests occur in this way if the same transfer class is applied to all transfer requests. If different transfer classes are applied to pending transfer requests, transaction requests alternate between transfer requests on a round-robin basis.

For example, if four transfer requests are pending with the following characteristics:

Transfer Identifier	Transfer Class	Number of Transactions Required
TA	0	2
TB	0	1
TC	1	2
TD	2	3

The sequence of transaction requests from the BMC to the DLMI with these pending transfer requests are:

Transfer Identifier	Transaction Number	Transfer Complete?
TA	0	No
TC	0	No
TD	0	No
TA	1	Yes
TC	1	Yes
TD	1	No
TB	0	Yes
TD	2	Yes

9.5. BMC Per-Channel Registers

A coprocessor 3 register set exists for each channel. Per-channel registers are implemented using the coprocessor 3 general registers. General registers are read using the mfc3 instruction, and written using the mtc3 instruction. The contents of the general registers persist until a new value is written. The 0 fields in these registers are ignored on write and are 0 on read. For compatibility with future LX8380 versions, the 0 fields should be written with 0.

BMC_CBUSADR (R16)

31 - 0
cbusAdr

Field	Description	R/W	Reset
cbusAdr	CBUS transfer physical starting address	R/W	0

BMC_DMEMADR (R17)

31 - 18	17 - 0
0000_0000_0000_00	dmemAdr

Field	Description	R/W	Reset
dmemAdr	DMEM transfer physical starting address. Only the bits used by the configured DMEM size are required (e.g. 64K DMEM requires bits 15:0). Other bits are don't care.	R/W	0

BMC_XFERLEN (R18)

31 - 19	18 - 0
0000_0000_0000_0	xferLen

Field	Description	R/W	Reset
xferLen	Transfer length. 1 - 262,144 bytes (256K) 0 = no operation, > 262,144 undefined	R/W	0

BMC_PARAM (R19)

31 - 24	23 - 20	19 - 18	17	16
0000_0000	xferType	00	intEnable	bmcPriority

15 - 8	7 - 4	3	2	1	0
0000_0000	xferClass	noCbusInc	noLine	noDword	noWord

Field	Description	R/W	Reset
xferType	0001 = read transfer 0010 = write transfer other encoding = reserved	R/W	0
intEnable	0 = disable interrupt; 1 = enable interrupt	R/W	0
bmcPriority	0 = CPU Priority; 1 = BMC priority	R/W	0
xferClass	transfer class (0 - 15)	R/W	0
noCbusInc	0 = increment CBUS address; 1 = don't increment CBUS address	R/W	0
noLine	0 = allow line transactions; 1 = disable line transactions	R/W	0
noDword	0 = allow dword transactions; 1 = disable dword transactions	R/W	0
noWord	0 = allow word transactions; 1 = disable word transactions	R/W	0

BMC_CMD (R20)

31	30 - 3	2 - 0
bmcBusy	000_0000_0000_0000_0000_0000_0000_0	command

Field	Description	R/W	Reset
bmcbusy	0 = no transfer in progress; 1 = transfer in progress Set to 1 when a transfer is initiated for this channel. Set to 0 when the transfer completes, or when stop transfer command is issued	R	0
command	000 clear done 001 start transfer, set BMC_DONEVEC's bit for this channel = 0, set bmcBusy = 1 010 stop transfer, set BMC_DONEVEC's bit for this channel = 1, set bmcBusy = 0 others reserved.	R/W	0

9.6. BMC Global Registers

Global registers apply to all channels. They are used to manage interrupts and the per-channel registers. Global registers are implemented using the coprocessor 3 control registers. Control registers are read using the cfc3 instruction, and written using the ctc3 instruction. The 0 fields in these registers are ignored on write and are 0 on read. For compatibility with future LX8380 versions, the 0 fields should be written with 0.

BMC_REGSET (C16)

31 - 9	8	7 - 4	3 - 0
0000_0000_0000_0000_0000_000	useActiveSet	0000	activeRegSet

Field	Description	R/W	Reset
useActiveSet	0 = don't use activeRegSet, instead use active context 1 = use activeRegSet	R/W	0
activeRegSet	Channel whose register set is the target of coprocessor 3 reads and writes.	R/W	0

BMC_DONEVEC (C17)

31 - 16	15 - 0
0000_0000_0000_0000	doneVec

Field	Description	R/W	Reset
doneVec	Bit vector corresponding to each channel. In any bit position: 0 = transfer not complete 1 = transfer complete A channel's bit is set = 1 when its transfer completes, or a stop transfer command is issued. The bit is set = 0 when the clear done or start transfer command is issued.	R	0

BMC_INTLOW (C18)

31	30 - 4	3 - 0
intPending	000_0000_0000_0000_0000_0000_0000	lowestInt

Field	Description	R/W	Reset
intPending	0 = no interrupt pending; 1 = interrupt pending	R	0
lowestInt	Encoded value of lowest numbered channel with a pending interrupt	R	0

BMC_INTENVEC (C19)

31 - 16	15 - 0
0000_0000_0000_0000	intEnVec

Field	Description	R/W	Reset
intEnVec	Mirror of the BMC_PARAM intEnable bit for each channel.	R/W	0

9.7. Per-Channel Register Set Selection

The BMC supports up to 16 channels. Each channel has its own register set used for specifying and controlling BMC transfers. Two techniques are provided for selecting which register set is accessed by software during coprocessor 3 reads and writes:

1. Explicit register set selection: The register set accessed may be selected by placing its value in BMC_REGSET[activeRegSet], and setting BMC_REGSET[useActiveSet] = 1. This mode allows access to any channel's register set.

2. Automatic register set selection. The register set accessed is selected automatically based on what context is active. This mode is selected by setting BMC_REGSET[useActiveSet] = 0. When using this mode, only one transfer request may be pending per context.

9.8. Transfer Completion

A transfer associated with a particular channel must complete before another transfer is requested using that channel. Software may check for transfer completion using the following techniques:

1. Polling. Software can poll if a transfer is in progress (BMC_CMD[bmcBusy]) or if a transfer is done (BMC_DONEVEC). The transfer done bit remains asserted until cleared.

2. Interrupt. The BMC interrupt line (BMC_INT_R_N) is an output of the LX8380 processor. The customer may connect this signal to any of the INTREQ_N[15:2] inputs. In the Lexra test-bed environment, the BMC interrupt is connected to INTREQ_N[12].

   The BMC interrupt is the logical OR of the BMC_DONEVEC bits for every channel ANDed with its associated interrupt enable bit. The enable bit is available in the per-channel BMC_PARAM register, or simultaneously for every channel in the BMC_INTENVEC register. A Verilog representation of the BMC interrupt signal is:

```
interrupt = 0;
for (i = 0; i < number_of_channels; i = i + 1)
   interrupt = (BMC_DONEVEC[i] & BMC_INTENVEC[i]) | interrupt;
```

The interrupt handler must clear the appropriate BMC_DONEVEC bits when interrupt processing is complete. Five instructions must pass after the write that clears BMC_DONEVEC before interrupts are re-enabled.

3. Wait-Event. To use the Wait-Event mechanism to activate a context after transfer completion, software uses a CSW instruction with Wait-Event bit 2 set (see the Example Transfer Flow section). The context will not reactivate until Wait-Event bit 2 clears. The BMC signals transfer completion by clearing Wait-Event bit 2 for the context that requested the transfer, allowing that context to reactivate.

Some write transactions may be pending in the CBI write buffer when write transfer completion is signalled.

9.9. CPU-BMC arbitration

When a one port DMEM interface is configured, requests for DMEM access from the CPU and BMC are arbitrated. Arbitration applies to DMEM accesses related to both read and write BMC transfers. Each channel's register set has a control bit (BMC_PARAM[bmcPriority]) which is used to select the priority of that channel's BMC request relative to the CPU.

Arbitration occurs on a per transaction basis. This implies that multiple arbitration events are required to complete a single transfer.

9.10. Software Responsibility for Transfer Requests

The data cache is never accessed as a result of a BMC transaction. If the CBUS address used for a transaction hits an address that is resident in the data cache, the data cache and main memory become incoherent as result of the transfer. It is software's responsibility to manage cache coherency.

Software must also ensure that no portion of the requested transfer exceeds the limits of physically configured DMEM. The BMC provides no checking for inconsistent transfer specifications.

9.11. Example Transfer Flow

The following BMC transfer flow demonstrates the use of Wait-Event bits to signal transfer completion.

1. For write transfers, software moves the write data into DMEM.

2. Software selects the mode for channel selection by using a **ctc3** instruction to load BMC_REGSET.

3. Software sets up transfer characteristics by using **mtc3** instructions to load BMC general registers.

4. Software starts the transfer by using a **mtc3** instruction to load a command in BMC_CMD. A context switch (via the CSW instruction) is normally performed so another thread can continue execution while the transfer is in progress. The following code sequence is typical:

```
csw     r1              # r1 contains 0x04000000 to set wait-event
                        #       bit 2 in this context's CXSTATUS
mtc3    r2, BMC_CMD     # start BMC transfer in csw delay slot
                        #       r2 contains start command
```

5. The BMC makes transaction requests to the DLMI, waiting for an acknowledgment before proceeding to the next request. While the BMC makes these requests, other threads continue execution.

6. The DLMI services each request by issuing the appropriate transactions, and returning an acknowledgment to the BMC when it is able to accept a new request.

7. When the final transaction request has been acknowledged, the BMC clears Wait-Event bit 2 for the requesting context, allowing it to resume execution. If the completed transfer is a write transfer, software must cause a write buffer flush (technique TBD) before attempting to access data.

The following BMC transfer flow demonstrates the use of interrupts to signal transfer completion.

1. For write transfers, software moves the write data into DMEM.

2. Software selects the mode for channel selection by using a **ctc3** instruction to load BMC_REGSET.

3. Software sets up transfer characteristics by using **mtc3** instructions to load BMC general registers. The BMC_PARAM[intEnable] bit must be asserted to allow interrupts.

4. Software starts the transfer by using a **mtc3** instruction to load a command in BMC_CMD.

5. The BMC makes transaction requests to the DLMI, waiting for an acknowledgment before proceeding to the next request. While the BMC makes these requests, software continues execution.

6. The DLMI services each request by issuing the appropriate transactions, and returning an acknowledgment to the BMC when it is able to accept a new request.

7. When the final transaction request has been acknowledged, the BMC causes the BMC_INT_R_N signal to assert. This leads to a hardware interrupt. The interrupt handler software can determine what channel caused the interrupt by examining BMC_DONEVEC and BMC_INTENVEC. If more than one channel is causing the interrupt, BMC_INTLOW provides a quick method for identifying the lowest-numbered interrupting channel. The interrupt handler must clear the interrupting channel's request by loading a clear done command in BMC_CMD.

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 LX8380's CBUS or LBUS, bypassing internal caches and memories. SoC designers need only provide package connections to the LX8380'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 LX8380 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 LX8380 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 LX8380. Information on these products is available at the following web sites.

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

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

LX8380 EJTAG implements all required features of version 2.0.0 of the EJTAG specification, including:

- The LX8380 may access debug host resources via addressing of probe memory space.
- Debug host can DMA directly to or from devices attached to the LX8380's system bus.
- Hardware breakpoints may be installed on internal LX8380 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 43: 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 44: 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 45: 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 LX8380 EJTAG includes support for real-time Program Counter (PC) Trace. When in PC Trace mode, the LX8380 serially outputs a new value of the program counter whenever there is a change in the PC (i.e. a context switch, 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 LX8380: 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 LX8380'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 LX8380's SYSCLK, up to N PCST values are output simultaneously.

Changes in program flow caused by a context-switch are shown by the JMP PCST code. In addition, the PCST codes for the context-switch (JMP) and its branch-delay slot (SEQ) are switched so that the branch-delay slot will be shown first, and any subsequent delay due to no context being ready is shown by the STL (stall) PCST code. This causes the following PCST output:

Case1: Context Switch to immediate dispatch of another context:

cntx		PCST
1	foo1a	# SEQ
1	csw	# SEQ
1	foo1b	# JMP
2	foo2a	# SEQ/JMP/EXP
2	foo2b	# ...

Case2: Context Switch with no ready context

cntx		PCST
1	foo1a	# SEQ
1	csw	# SEQ
1	foo1b	# STL
*	nvld	# STL
...		# JMP
2	foo2a	# SEQ/JMP/EXP
2	foo2b	# SEQ

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 46: 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 47: 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 LX8380 from all other exceptions, the LX8380 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

1. Based on EPI Interface Specifications for MAJIC^{TM} and MAJIC$^{PLUS\ TM}$

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 LX8380 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.

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.

10.3. Data Break Exceptions for LX8380

The existing EJTAG data match architecture does not allow matches for some of the transaction types in the LX8380. This is described in more detail below.

10.3.1. Data Break Data Matches on LBus Split Transactions

Data break matches (address and/or data) on LBus split transactions are not supported. Such transactions are generated by any context-switch instruction (*.CSW instructions).

10.3.2. Data Breaks on Write Descriptor Accesses

Data breaks on the address or data of write descriptor (all WD.* instructions) accesses are not supported.

10.3.3. Support for the Load-Twin Instruction

Data matches on the Load-Twin instruction are supported. The 32-bit entry in the Data Value Break register will be compared to both halves of the 64-bit data returned by this instruction. Therefore any masking of the data byte lanes must be copied from bits 7:4 (Byte Lane Mask[3:0]) to bits 11:8 in the Data Break Control register to ensure the same mask is applied across both words returned.

Appendix A.Instruction Formats

This appendix documents the LX8380 instruction encodings that are not included in the standard MIPS-I (R2000/R3000) instruction set.

A.1. Major Opcodes

Table 48: 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 49: 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.2. LEXOP2 Instructions

Table 50: 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
LW.CSW	LEXOP2	base	rt	displacement/4	LWC
LT.CSW	LEXOP2	base	rt-even, 0	displacement/8	LTC
LQ.CSW	LEXOP2	base	rt-quad, 00	displacement/16	LQC
	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.
rt-quad — Selects general register quad r0/r1/r2/r3 ... r28/r29/r30/r31.
displacement — Signed 2s-complement number in bytes.

Table 51: LEXOP2 Write Descriptor Instruction Formats

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**LEXOP2 011 110**	**rs**	**rt**	**rd**	**deviceID**	**Lexra SUBOP**
WD	LEXOP2	rs	rt	0	deviceID	WD
WD.CSW	LEXOP2	rs	rt	0	deviceID	WDC
WDLW.CSW	LEXOP2	rs	rt	rd	deviceID	WDLWC
WDLT.CSW	LEXOP2	rs	rt	rd-even,0	deviceID	WDLTC
WDLQ.CSW	LEXOP2	rs	rt	rd-quad,00	deviceID	WDLQC
	6	5	5	5	5	6

rs, rt, rd — Selects general register r0 - r31.
rd-even — Selects general register even-odd pair r0/r1, r2/r3 ... r30/r31.
rt-quad — Selects general register quad r0/r1/r2/r3 ... r28/r29/r30/r31.
deviceID — indicates bits 7:3 of system device address.

Table 52: LEXOP2 Context, Checksum and Bit Field Formats

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**LEXOP2 011 110**	**rs**	**rt**	**rd**	**0**	**Lexra SUBOP**
MYCX	LEXOP2	0	0	rd	0	MYCX
POSTCX	LEXOP2	rs	rt	0	0	POSTCX
CSW	LEXOP2	rs	0	0	0	CSW
EXTIV	LEXOP2	rs	rt	rd	0	EXTIV
INSV	LEXOP2	rs	rt	rd	0	INSV
ACS2	LEXOP2	rs	rt	rd	0	ACS2
MSB	LEXOP2	rs	rt	rd	0	MSB
JOR	LEXOP2	rs	rt	0	0	JOR
	6	5	5	5	5	6

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**LEXOP2 011 110**	**rs**	**rt**	**width**	**keysize/ offset**	**Lexra SUBOP**
SETI	LEXOP2	rs	rt	width	offset	SETI
CLRI	LEXOP2	rs	rt	width	offset	CLRI
EXTII	LEXOP2	width	rt	rd	offset	EXTII
INSI	LEXOP2	rs	rt	rd	offset	INSI
HASH	LEXOP2	rs	0	rd	keysize	HASH
	6	5	5	5	5	6

rs, rt, rd	Selects general register r0 - r31.
width	a 5-bit encoding of the width parameter modulo 32. (i.e. the value 32 is represented as 0).
offset	a 5-bit encoding of the offset parameter in the range 0-31.
keysize	a 5-bit encoding of the keysize parameter in the range 4-24.

Table 53: Cross Context Move Format

	31 26	25 21	20 16	15 11	10 6	5 0
Assembler Mnemonic	**LEXOP2 011 110**	**0**	**rt/gt/ct**	**rd/gd/cd**	**0**	**Lexra SUBOP**
MFCXG	LEXOP2	0	gt	rd	0	MFCXG
MTCXG	LEXOP2	0	rt	gd	0	MTCXG
MFCXC	LEXOP2	0	ct	rd	0	MFCXC
MTCXC	LEXOP2	0	rt	cd	0	MTCXC
	6	5	5	5	5	6

rt, rd — Selects general register r0 - r31 in the current context.
gt, gd — Selects general register r0 - r31 in the context specified by MOVECX.
ct, cd — Selects context register in the context specified by MOVECX:
00000 = CXSTATUS
00001 = CXPC
others = reserved

Table 54: LEXOP2 Subop Bit Encodings

Inst[5:3] \ Inst[2:0]	0	1	2	3	4	5	6	7
0				HASH	SETI	ACS2	INSV	INSI
1	JOR			MSB	CLRI		EXTIV	EXTII
2								
3								
4	MYCX				MFCXG	MTCXG		
5	POSTCX				MFCXC	MTCXC		
6	CSW	LQC	WDC	WDLQC	LTC	LWC	WDLTC	WDLWC
7			WD		LTW			

A.3. COP0 Instructions

Table 55: 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)
00100 MOVECX
00101 reserved
0011x reserved
01xxx reserved
1xxxx reserved

Table 56: 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.4. SPECIAL Instructions

Table 57: 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 58: 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

Table 59: 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**
SDBBP	SPECIAL2	00000	00000	00000	00000	SDBBP
	6	5	5	5	5	6

Table 60: SPECIAL2 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								SDBBP

Appendix B.Lconfig Forms

B.1. Configuration Options for the LX8380 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 61: 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
CONTEXTS	Number of contexts (threads) in the processor
COP1	coprocessor interface 1
COP2	coprocessor interface 2
COP3	coprocessor interface 3 (BMC)
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

***Lconfig* Option**	**Description**
LBC_SYNC_MODE	LBC synchronous/asynchronous selection
LINE_SIZE	cache line size, in words
DMEM_WIDTH	local scratch pad data memory width
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
MMU_PAGE_SIZE	memory page size
PC_TRACE	EJTAG PC trace pins
PRODUCT	Lexra Processor name
REGFILE_TECH	register file technology
RESET_BUFFERS	reset buffers at top-level module
RESET_TYPE	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
SYSTEM_INTERFACE	system bus interface type
TECHNOLOGY	identifies target technology
THREAD_SCHEDULER	location of thread scheduler
TLB_ENTRIES	number of entries in Translation Lookaside Buffer
WDESC_ADDR	Write Descriptor upper address bits
WRITETHROUGH_RANGE	writethrough range for writeback data cache

Appendix C.Port Descriptions

Table 62 shows the possible port connections for the top level module of the LX8380 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 62: LX8380 Processor Port Summary

Port Name	I/O	Timing	Description
Clocking, Reset, Interrupts and Control			
SYSCLK	input	(3)	Processor clock.
BUSCLK	input	(3)	Bus clock, if processor is configured with async LBC.
ResetN	input	10%	Warm reset (or reset "button"), active low.
CResetN	input	10%	Cold reset (or power on), active low.

Port Name	I/O	Timing	Description
RESET_D1_R_N	input	30%	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.
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_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_PROCNUM[7:0]	input	(4)	Strapped with processor number.
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.

Port Name	I/O	Timing	Description
CFG_IWTOP[`23:10]	input	30%	Strapped with IMEM top address configuration value.
CFG_DWDISW	input	(4)	Strap to one to disable processor DMEM writes. Must be zero for LX8380.
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.

Port Name	I/O	Timing	Description
Data RAM DMA Access			
DMADW_RCLK	input	(3)	Data RAM DMA clock.
DMADW_DATAINDEX[17:4]	input	(5)	Data RAM DMA address (max size).
DMADW_DATARD[63:0]	output	(5)	Data RAM DMA read data (128-bit interface is optional).
DMADW_DATAWR[63:0]	input	(5)	Data RAM DMA write data (128-bit interface is optional).
DMADW_DATACS	input	(5)	Data RAM DMA chip select.
DMADW_DATACSN	input	(5)	Data RAM DMA chip select, active low.
DMADW_DATARE	input	(5)	Data RAM DMA read enable.
DMADW_DATAREN	input	(5)	Data RAM DMA read enable, active low.
DMADW_DATAWE[<k>:0]	input	(5)	Data RAM DMA write enable, where <k> is 3 for word write granularity, 15 for byte write granularity.
DMADW_DATAWEN[<k>:0]	input	(5)	Data RAM DMA write enable, active low, where <k> is 3 for word write granularity, 15 for byte write granularity.
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.
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.
LGTidO[15:0]	output	(2), 20%	LBC global thread ID.
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.

Port Name	I/O	Timing	Description
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.
Czrd_cntx[2:0]	output	40%	Cop read context number
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_cntx[2:0]	output	30%	Cop write context number
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.
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.)

Port Name	I/O	Timing	Description
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_YSPLTO	output	20%	1=Split, 0=normal transaction.
CBUS_YLTIDO[3:0]	output	20%	Local thread ID
CBUS_YUCO	output	20%	1=uncached, 0=cached access.
CBUS_YSRCO[3:0]	output	20%	transaction source (within LX8380): 4'b0001 Instruction Cache 4'b0010 Data Cache or EJTAG DMA write 4'b0100 EJTAG DMA read 4'b1000 BMC
CBUS_YDBUSYO	output	20%	1 - LX8380 is not ready to receive data for a Data Read or Data Split Read. Any return read data with VALTYPE of Data Read or Data Split Read will be ignored by the LX8380. External logic must hold such data CBUS_YDBUSYO is de-asserted. 0 - LX8380 is ready to receive data.
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_YLTIDI[3:0]	input	80%	Context associated with 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 BMC
CBUS_YSPLTSZI[2:0]	input	80%	Size of split Read Data beat: 3'b000 - 1 byte 3'b001 - 2 bytes 3'b011 - 1 word 3'b100 - 2 words
CBUS_YIDLEI	input	80%	Indicates external CBUS_Y device has no pending read or write transactions.

Port Name	I/O	Timing	Description
Event Control and Thread Scheduling			
EXT_CLEARWTEVNT_R [<n>*8-1:0]	input	30%	Clear status wait event bits, where <n> is the number of contexts.
CX_STUSTHWAIT_R [<n>-1:0]	output	30%	Bits set to one indicate which contexts are waiting for events, where <n> is the number of contexts.
CX_THREADACTV_R[<n>-1:0]	output	30%	A bit set one indicates which context (if any) is active, where <n> is the number of contexts.
EXT_NXTCNTX_P_R[2:0]	input	30%	External Scheduler Next Context.
EXT_NEXTCNTXRDY_P_R	input	30%	External Scheduler Next Context is ready.
CX_STUSTHPRIO_R[<n>*3-1:0]	output	30%	Thread priority status.

Appendix D.Pipeline Stalls

This appendix documents the stall conditions that may arise in the LX8380.

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 63: 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.

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.

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. MOVECX Stall

In addition to its MVCz stall, a MTLXC0 instruction to the MOVECX register is followed by two more issue stalls, for a total of four issue stalls.

D.9. 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.10. 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.11. 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.12. 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.13. 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.14. 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.15. 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.16. 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