LX8000 Data Sheet

Lexra, Inc.

Release 1.9

March 29, 2001

LX8000 Data Sheet Revision 1.3, for RTL Release 1.9.

Table of Contents

List of Figures

List of Tables

1. LX8000 Product Overview

1.1. Introduction

The LX8000 is based on Lexra's LX4189 processor, a complete MIPS R3000-class processor subsystem developed for ease of integration. (See Figure 1 on page 11.) The major subsystems are: the CPU core, Local Memory Interfaces (LMI) and LBus Controller (LBC). The technology includes optional interfaces to customer-defined Coprocessors (CI[1-3]) and optional customer extensions to the MIPS ISA (Custom Engine). The local instruction memories and data memories may include fixed RAM and/or cache; the sizes are configurable. The figure also highlights the LX8000 multi-context register file to support fast context switching. Additional LX8000 extensions include new bit-field operations for efficient packet header processing, and a dedicated data RAM port that the application may use for background data transfers.

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. LX8000 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. LX8000 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 LX8000 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, LX8000 has extended the MIPS 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 LX8000'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.

Because the LX8000 packet processor executes the MIPS I instruction set[1], a wide variety of third party software tools are available including compilers, operating systems, debuggers and in-circuit emulators. Lexra also supplies assembler extensions and a cycle accurate Instruction Set Simulator (ISS). Programmers may use "off-the-shelf" C compilers for initial coding, then replace performance critical code with optimized

1. Unaligned load and store instructions are not supported in hardware or software.

assembler code.

1.2. Key Features

- **Complete Packet Processor Subsystem**
  - Executes MIPS I ISA (except unaligned loads, stores)
  - Extensive third-party tool support
  - High-performance 6-stage pipeline,
  - Local instruction memory and/or cache, configurable sizes
  - Local data memory and/or cache, configurable sizes
  - Memory interface logic included
  - System bus controller.
  - Split read transactions to system bus devices.
  - Optional customer-defined coprocessors
  - Optional customer-defined instruction extensions
    support EJTAG Draft 2.0 with extensions for multi-thread debugging
- **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.
- **Bit-Field Instructions**
  - Single-cycle extract, set, clear.
  - Two-cycle extract-and-insert, with source fields that may span two registers.
  - Dual 16-bit ones complement add for checksum.
- **Portable RTL Model**
  - Available as a synthesizable RTL.
  - Portable to any 0.25µm, 0.18µm or 0.15µm
    logic and SRAM process.
  - Foundry partners include IBM, TSMC, and UMC.
- **Optional Hard Macro Model**
  - Processor with 4 contexts, 16KB IMEM, 16KB DMEM.
  - TSMC 0.15µm. results (typical process, worst case operating conditions):
    Clock: 450 MHz
    Area: 4.7 mm^2
    Power: 210 mW
- **Easy ASIC Design**
  - Single phase clocking.
  - Fully synchronous design.
  - Easy to interface system bus protocol.
  - Supports popular EDA tools.

- **Easy RTL Customization**
  - User-configurable local memory, reset method, clock distribution.
  - User-configurable EJTAG breakpoints.
  - Over 30 other configuration options.
  - Interfaces for adding application-specific instructions.
- **EJTAG Debug**
  - Optional extension the EJTAG 2.0.0 specification
  - Supports multi-context environment for on-chip debug.
- **Development Tools**
  - Available from third party suppliers supporting the MIPS architecture
  - Includes industry leaders Green Hills Software, Embedded Performance Inc., and Wind River Systems.

1.3. LX8000 Processor Overview

The LX8000 is a RISC processor that executes the MIPS-I instruction set[1] along with Lexra's instruction set extensions. However, the clocking, pipeline structure, pin-out, and memory interfaces have all been designed by Lexra to reflect system-on-silicon design needs, deep sub-micron process technology, as well as design methodology advances.

The figure below shows the structure of the LX8000 processor.

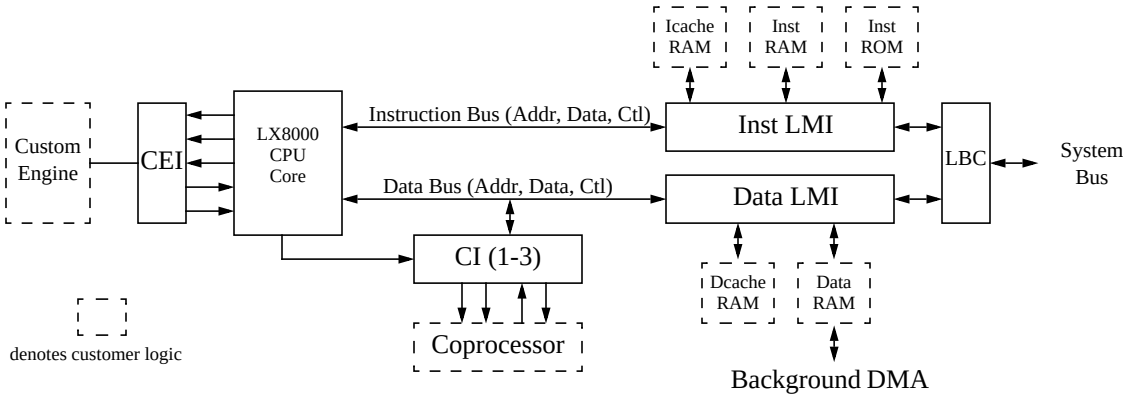


Figure 1: LX8000 Processor Overview

MIPS ISA Execution. The LX8000 supports the MIPS I programming model. Two source operands can be supplied and one destination update performed per cycle. The second operand is either a register or 16-bit immediate. The instruction set includes a wide selection of ALU operations executed by the RALU, Lexra's proprietary register based ALU. The RALU also generates memory addresses for 8-bit, 16-bit, and 32-bit register loads from (stores to) memory by adding a register base to an immediate offset. Branches are based on comparisons between registers, rather than flags, and are therefore easy to relocate. Optional links following jump or branch instructions assist with subroutine programming.

The MIPS unaligned load and store instructions are not supported, because they represent poor price/

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

performance trade-off for embedded applications. Their absence does not affect the software programming model.

ISA Extensions for Network Processing. Lexra has added 32 new instructions to the LX8000 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. LX8000 instructions are executed by a six-stage pipeline that has been designed so that all transactions internal to the LX8000, as well as at the interfaces, occur on the positive edge of the processor clock. Two-phase clocks are not used.

Context Switching. The LX8000 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 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.

Exception Handling. The MIPS R3000 exception handling model is supported. Exceptions include both instruction-synchronous *traps* as well as hardware and software *interrupts*. The 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 CAUSE register. A user program located at the exception vector identifies the cause of the exception, and transfers control to the application-specific handler. In the event of an address error exception, the BADVADDR holds the failing address.

Coprocessor Operations. The LX8000 supports 32-bit Coprocessor operations. These include moves to and from the Coprocessor general registers and control registers (MTCz, MFCz, CTCz, CFCz), Coprocessor loads and stores (LWCz, SWCz) and branches based on Coprocessor condition flags (BCzT, BCzF). The Lexra-supplied Coprocessor Interface can support Coprocessor operations in a single cycle, without pipeline stalls.

LX8000 provides excellent price/performance and time-to-market. There are two main approaches which Lexra has taken 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 hardcore (called SmoothCore™) for popular pure-play foundries.

Section 1.4 describes the building blocks, and Section 1.5 describes the deliverable models.

1.4. System Level Building Blocks

The LX8000 processor is designed to easily fit into different target applications. It provides the following building blocks.

- A simple memory management unit (SMMU).
- An optimized Custom Engine Interface (CEI).
- Up to three Coprocessor Interfaces (CI).
- A flexible Local Memory Interface (LMI) that supports instruction cache, instruction RAM, instruction ROM, data cache and data RAM.
- A Lexra Bus Controller (LBC) to connect peripheral devices and secondary memories to the processor's own local buses.

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

1.4.1. SMMU

The LX8000 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.

1.4.2. Local Memory Interface

The LX8000'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 a two-way set associative instruction cache interface, and a direct-mapped write-through data cache interface. 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. Local instruction and data memories can also be mapped to fixed regions of the physical address space, and include non-volatile memory (such as ROM, flash, or EPROM).

1.4.3. Coprocessor Interface

Lexra supplies an optional Coprocessor Interface (CI) for applications requiring this functionality. Up to three CIs may be implemented in one design. The Coprocessor Interface "eavesdrops" on the Instruction bus. If a Coprocessor load (LWCz) or "move to" (MTCz, CTCz) is decoded, data is passed over the Data Bus into a CI register, then supplied to the designer-defined Coprocessor. Similarly, if a Coprocessor store (SWCz) or "move from" (MFCz, CFCz) 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 design interface includes a data bus, five-bit address, and independent read and write selects for Coprocessor registers and control registers. The LX8000 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 Sequencer for testing in branch instructions.

1.4.4. Custom Engine Interface

The LX8000 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 LX8000. To support multi-cycle operations, a stall input is included in the interface.

1.4.5. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the interface between the LX8000 and the outside world, which includes DRAM and various peripherals. It is a non-multiplexed, non-pipelined, and non-parity checked bus to provide the easiest bus protocol for design integration. On the processor side, the LBC provides a write-buffer of configurable depth to support the write-through cache, as well as the control for byte and half-word transfers. On the peripheral side, the LBC is designed to easily interface to industry standard bus protocols, such as PCI, USB, and FireWire.

The LBC can run at any speed from 33 MHz, up to the speed of the LX8000 processor core in both the RTL core and SmoothCore.

1.4.6. Building Block Integration

The LX8000 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 include scan insertion, which allows ATPG testing of the entire LX8000 core.

1.5. RTL Core & SmoothCore

Lexra delivers LX8000 as RTL Core and SmoothCore.

RTL Core: For full ASIC designs, the RTL 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 LX8000.

SmoothCore: For COT designs that are manufactured at popular foundries such as IBM, TSMC, and UMC, a SmoothCore port is the quickest, lowest cost, and best performance choice. In this case, LX8000 has been fully implemented and verified as a hard macro. All data path, register file, and interface optimizations have been performed to ensure the smallest die size and fastest performance possible. Furthermore, there is a scan based test pattern that provides excellent fault coverage during manufacturing tests.

1.6. 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. LX8000 Architecture

2.1. Hardware Architecture

2.1.1. Module Partitioning

The LX8000 processor core includes two major blocks: the RALU (register file and ALU) and the CP0 (Control Processor). The RALU performs ALU operations and generates data addresses while CP0 includes instruction address sequencing, exception processing, and product specific mode control. The RALU and CP0 are loosely-coupled and include their own independent instruction decoders.

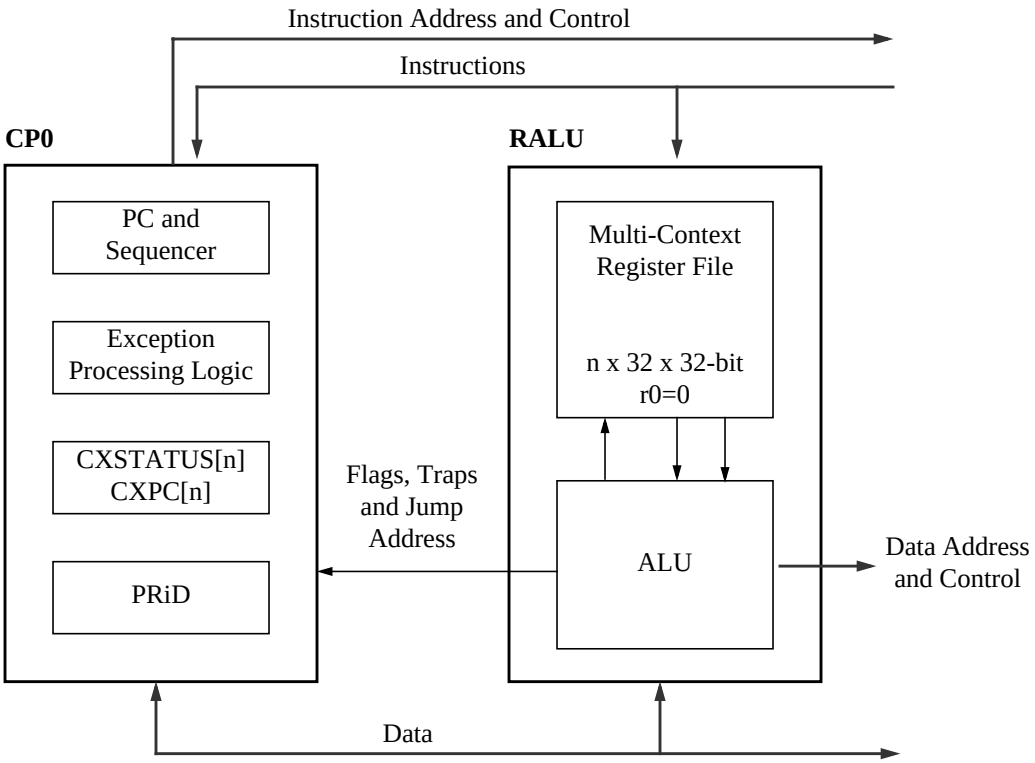


Figure 2: Processor Core Module Partitioning

2.1.2. Six Stage Pipeline

The LX8000 has a six stage pipeline:

Stage 1	I	Instruction fetch
Stage 2	D	Decode
Stage 3	S	Source fetch (register file read)
Stage 4	E	Execution and address generation
Stage 5	M	Memory data select (read data cache store and tags)
Stage 6	W	Write back to register file

The six stage pipeline provides a complete processor cycle for the instruction memory, providing ease of use integrating for allowing use of larger and set-associative memories without degrading cycle time. The six pipeline stages allow the processor clock speed to scale with current silicon processes.

As a result of the D-Stage, a two cycle penalty is incurred on branch prediction failure vs. the one-cycle penalty in the LX4180 five stage pipeline. However, the LX8000's conditional move instructions can be used to avoid any wasted cycles in the control of real-time critical loops.

2.2. RALU Data Path

The LX8000 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 LX8000, 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 LX8000 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 one of three optional application-specific Coprocessors The general purpose registers may also be loaded from the Coprocessor registers. These transfers occur over the Data Bus, similar to data memory loads and stores.

2.3. 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 LX8000 throughput by several percent. However, the net effect of the LX8000'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) 0x0000c701 for LX8000
CCTL	20	Instruction and data memory control

EPC, STATUS, CAUSE, and BADVADDR are described further in the Section 3.4. PRID is a read-only register that allows the customer's software to identify the specific version of the LX8000 that has been implemented in their product. The CCTL register is a Lexra defined CP0 register used to control the instruction and data memories, as described in Section 5.2, Cache Control Register: CCTL.

The contents of the above registers can be transferred to and from the RALU's general-purpose register file using CP0 operations. (Unlike registers located in Coprocessors 1-3, they cannot be loaded or stored directly to data memory.)

2.4. High-Performance Context Switch

The LX8000 CPU incorporates multiple, independent register sets called *contexts*. As a result, execution can switch between independent 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 LX8000 provides a zero-overhead context switch. That is, an instruction can be executed for *some* context in every cycle.

2.4.1. New Context Registers

The number of contexts is customer-defined using Lexra's *lconfig* utility. One to eight contexts are supported by the LX8000 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.

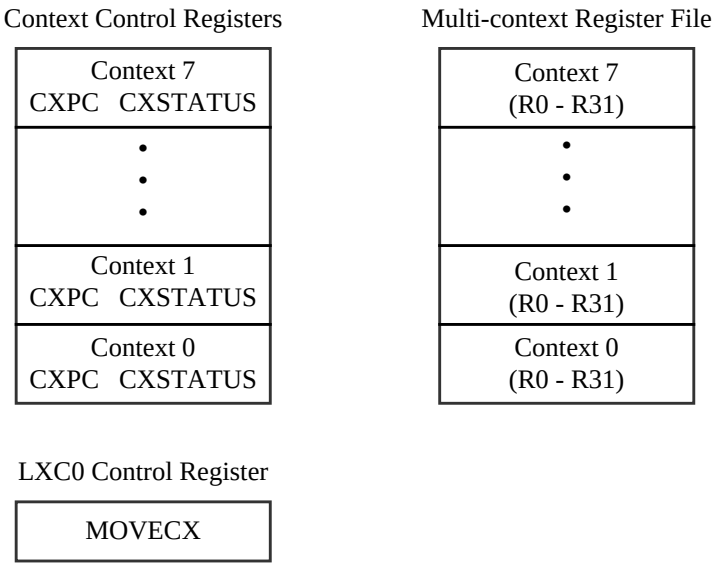


Figure 3: Context Associated Registers

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 thread. The 16-bit CXSTATUS register indicates whether the thread is waiting for data transfer or I/O events. CXSTATUS also permits program-assigned priority for thread re-activation.

The CXSTATUS register fields are identified in Table 3. 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 thread 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 thread). 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 thread is ready for execution.

The CXSTATUS WAIT-LOAD bit indicates that the associated thread 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 2.4.4 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 thread scheduling with up to eight priorities. An application specific thread scheduler can utilize thread priorities to fine tune the thread scheduling. See Section 2.4.4 for details of the thread scheduling hardware interface.

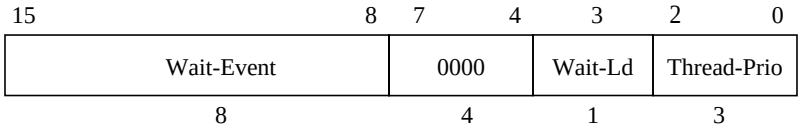


Table 3: Context Status Register Detail

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 hardware.
THREAD-PRIORITY	3	(Rd/Wr) Thread priority, for use by optional custom thread scheduler.

2.4.2. Reset

At reset,

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

The general registers are unaffected by reset.

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

2.4.3. 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 2.4.12, 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.

2.4.4. 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

2.4.5. 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.

2.4.6. 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 thread 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 thread unless the thread 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 thread 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.

2.4.7. 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.

2.4.8. 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.

2.4.9. 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:

	thread(i)		thread(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, thread(i)’s inst n+1 executes to completion. CXPCi stores the address of inst n+2; the address where thread(i) resumes when it is later re-activated. After inst n+1 is complete, the next instruction executed is inst m+1 in thread(j). Of course, thread(i) and thread(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.

2.4.10. New Thread Selection

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

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

The CPU logic required to implement the above next thread selection algorithm is pipelined. As a result, the next thread 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 thread selection will occur several cycles earlier and may not include a newly available thread. However, this is not a drawback because event completions for inactive threads are asynchronous to the current thread's program. The LX8000's internal thread scheduler (described in the following paragraphs) is pipelined such that if there is currently no active thread (all threads are have some wait bit set), it takes two cycles from the time that some thread has all of its Wait bits clear, until that thread's CXPC value is driven to the instruction RAM.

The LX8000 processor includes internal thread 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 LX8000's internal thread scheduler ignores the THREAD-PRIORITY field of the CXSTATUS register. It selects the next thread "fairly". A characteristic of this scheduler is that, if threads are performing similar types of activities over time, they experience similar selection rates and similar delays in selection when there are multiple threads ready for execution.

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

- Once a window of ready threads has been chosen, no other threads are added to this window.
- If a ready thread in the window subsequently has one of its Wait bits turned on, that thread is removed from the window. Since the window contains only inactive threads, this can only happen if the currently active thread executes a MTCXC to turn on another thread'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 thread from the window is selected for the next context switch. As each context-switch takes effect, the selected thread is removed from the window. The selection among the threads in the window is not architecturally defined and application software should not depend on any particular order. The current implementation selects the highest numbered thread 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 threads. (If there are none, this step repeats until there is at least one ready thread.) When a new non-empty window is obtained, the full cycle of this algorithm continues as described above.

Any thread that becomes ready will eventually be included in the next new window, and will be selected for execution. Therefore, this algorithm prevents a ready thread from being starved out of activation by other threads. The fairness of this algorithm results from the fact that threads 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 LX8000 scheduler and supply an application specific design that has access to the same per thread information as the standard scheduler. Such a scheduler may also examine other real time information that is outside the province of LX8000 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 4: 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 is active, where <n> is the number of contexts
EXT_NEXTCNTXRDY_P_R	input	1 if scheduler's next thread selection is valid
EXT_NEXTCNTX_P_R[2:0]	input	scheduler's next thread selection

Because the scheduler determines the thread 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 thread 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 thread 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 thread in case of a potential context switch.

To resolve the discrepancy between the end and beginning of the pipeline, CP0 inhibits a thread 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 threads 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 thread has been dispatched.

2.4.11. Example Context Switch for Coprocessor Operation

The following example illustrates how an unconditional context switch could be used to allow other threads to execute while a coprocessor performs a relatively long latency operation on behalf of a thread. 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 thread ready for dispatch. Since several threads 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 thread is made ready and
                    # eventually gets dispatched here
ctc2    r1, $1      # tell cop2 which context this is
mfc2    ...         # retrieve results
```

2.4.12. Program Access to New Registers

The new registers described in Section 2.4.1. CXPC, CXSTATUS, MOVECX, as well as the general registers of all contexts, are accessible under program control by the active thread.

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 thread 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 thread. If ct or cd is the CXPC of the currently active thread, the result of MTCXC or MFCXC is undefined.

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

```
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.)

2.4.13. Exceptions

The MIPS R3000 exception processing model is unchanged by LX8000, with one difference explained in the next paragraph. Following a program synchronous trap or an interrupt, the PC of the current thread 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.

LX8000 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.

Note that all threads share a common set of Coprocessor 0 registers including the exception processing registers listed in Table 2 on page 19, and the ESTATUS, ECAUSE and INTVEC registers described in Section 2.5.

2.5. Low-Overhead Prioritized Interrupts

The LX8000 includes eight new low-overhead hardware interrupt signals. These signals are compatible with the R3000 Exception Processing model and are useful for real-time applications.

These interrupts are supported with three new Lexra CP0 registers, ESTATUS, ECAUSE, and INTVEC, accessed with the new MTLXC0 and MFLXC0 variants of the MTC0 and MFC0 instructions. As with any COP0 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 new Lexra CP0 registers are ESTATUS (0), ECAUSE (1), and INTVEC (2), and are defined as follows:

ESTATUS (LX COP0 Reg 0) Read/Write

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

ECAUSE (LX COP0 Reg 1) Read-only

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

INTVEC (LX COP0 Reg 2) Read/Write

31 - 6	5 - 0
BASE	0

ESTATUS contains the new interrupt mask bits IM[15:8], which are reset to 0 so that none of the new interrupts will be activated, regardless of the global interrupt signal IEc. IP[15:8] for the new interrupt signals is located in ECAUSE and is read-only. These fields are similar to the IM and IP fields defined in the R3000 Exception Processing Model, except that the new interrupts are prioritized in hardware, and each have a dedicated exception vector.

IP[15] has the highest priority, while IP[8] has the lowest priority, however, all new interrupts are higher priority than IP[7:0]. The program defined BASE address for the exception vectors is located in INTVEC. The exception vector used for each prioritized interrupt is 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 5: Prioritized Interrupt Exception Vectors

Interrupt Number	Exception Vector
15	BASE || 111000
14	BASE || 110000
13	BASE || 101000
12	BASE || 100000
11	BASE || 011000
10	BASE || 010000
9	BASE || 001000
8	BASE || 000000

When a vectored interrupt causes an exception, all of the standard actions for an exception occur. These include updating the EPC register and certain subfields 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. LX8000 RISC Programming Model

This section describes the LX8000 Programming Model. Section 3.1, Summary of MIPS-I Instructions, contains a list summarizing all MIPS-I operations supported by the LX8000. These opcodes may be extended by the customer using Lexra's Custom Engine Interface (CEI). This capability is described in Section 3.2, Opcode Extension Using the Custom Engine Interface (CEI).

Section 3.3, Memory Management, describes the Simplified Memory Management Unit (SMMU) which is physically incorporated in the LX8000 LMI. The SMMU provides sufficient memory management capabilities for most embedded applications while ensuring execution of third-party MIPS software development tools.

The LX8000 supports the MIPS R3000 Exception Processing model, as described in Section 3.4, Exception Processing.

The LX8000 supports all MIPS-I Coprocessor operations. The customer can include one to three application-specific Coprocessors. Lexra provides a functional block called the Coprocessor Interface (CI) which allows the customer a simplified connection between their Coprocessor and the internal signals of the LX8000. The CI is described in Section 3.5, The Coprocessor Interface (CI).

Lexra's application specific instruction-set extensions are described in detail in 4, LX8000 Instruction Extensions.

3.1. Summary of MIPS-I Instructions

The LX8000 executes MIPS-I instructions as detailed in the tables below. To summarize, the LX8000 executes MIPS-I instructions with the following exclusions: the unaligned loads and stores (LWL, SWL, LWR, SWR) are not supported because they add significant silicon area for little benefit in most applications. The unaligned loads and stores execute as a NOP. This can cause code to execute incorrectly if the programmer expected these instructions to provide the unaligned load or store operations.

3.1.1. ALU Instructions

Table 6: ALU Instructions

Instruction		Description
ADD ADDU ADDI ADDIU	rD, rA, rB rD, rA, rB rD, rA, immediate rD, rA, immediate	rD <- rA + {rB, immediate} Add reg rA to either reg rB 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, rA, rB rD, rA, rB	rD <- rA - rB Subtract reg rB from reg rA. Result is stored in register rD. SUB can generate overflow trap. SUBU does not.
AND ANDI	rD, rA, rB rD, rA, immediate	rD <- rA & {rB, immediate} Logical *and* of reg rA with either reg rB or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
OR ORI	rD, rA, rB rD, rA, immediate	rD <- rA | {rB, immediate} Logical *or* of reg rA with either reg rB or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.

Instruction		Description
XOR XORI	rD, rA, rB rD, rA, immediate	rD <- rA ^ {rB, immediate} Logical *xor* of reg rA with either reg rB or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
NOR	rD, rA, rB	rD <- ~(rA | rB) Logical *nor* of reg rA with either reg rB or a zero-extended 16-bit immediate. Result is stored in reg rD.
LUI	rD, immediate	rD <- immediate || 16'(0) The 16-bit immediate is stored into the upper half of reg rD. The lower half is loaded with zeroes.
SLL SLLV	rD, rB, immediate rD, rB, rA	rD <- rB << {rA, immediate} Reg rB is left-shifted by 0-31. The shift amount is either the 5b immediate of the 5 lsb of rA. Result is store in reg rD.
SRL SRLV	rD, rB, immediate rD, rB, rA	rD <- rB >> {rA, immediate} Reg rB is right-shifted by 0-31. The unsigned shift amount is either the 5b immediate or the 5 lsb of rA. Result is stored in reg rD.
SRA SRAV	rD, rB, immediate rD, rB, rA	rD <- rB >>(a) {rA, immediate} Reg rB is arithmetic right-shifted by 0-31. The unsigned shift amount is either the 5b immediate or the 5 lsb of rA. Result is stored in reg rD.
SLT SLTU SLTI SLTIU	rD, rA, rB rD, rA, rB rD, rA, immediate rD, rA, immediate	rD <- 31'(0) || 1 if rA < {rB, immediate} else 0 If reg rA is less than {rB, 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	rD, offset(rA) rD, offset(rA) rD, offset(rA) rD, offset(rA) rD, offset(rA)	rD <- Memory[rA + offset] Reg rD is loaded from data memory. The memory address is computed as *base + offset*, where the base is reg rA 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 rD. rD cannot be referenced in the instruction following a load instruction.

Instruction	Description
SB rB, offset(rA) SH rB, offset(rA) SW rB, offset(rA)	rB -> Memory[rA + offset] Reg rB is stored to data memory. The memory address is computed as *base + offset*, where the base is reg rA 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 lsb of rB are stored. SH addresses are interpreted as halfword addresses to data memory; the 16 lsb of rB are stored.

3.1.3. Conditional Move Instructions

Table 8: Conditional Move Instructions

Instruction	Description
MOVZ rD, rS, rT	if rT = 0 rD <- rS else rD <- rD Conditional Move on Equal Zero 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	if rT != 0 rD <- rS else rD <- rD Conditional Move on Not Equal Zero 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 rA, rB, destination BNE rA, rB, destination	if COND pc <- (pc + 4) + 14'(destination[15]) || destination || 00 else pc <- (pc + 8) where COND = (rA = rB) for EQ, (rA ne rB) for NE, and destination is a 16-bit value. For BEQ, BNE the instruction after the branch (*delay slot*) is always executed.

Instruction		Description
BLEZ BGTZ	rA, destination rA, destination	if COND pc <- (pc + 4) + 14'(destination[15]) || destination || 00 else pc <- (pc + 8) where COND = (rA <= 0) for LE, (rA > 0) for GT, and destination is a 16-bit value For BLEZ, BGTZ the instruction after the branch (*delay slot*) is always executed.
BLTZ BGEZ	rA, destination rA, destination	if COND pc <- (pc + 4) + 14'(destination[15]) || destination || 00 else pc <- (pc + 8) where COND = (rA < 0) for LT, (rA >= 0) for GE, and destination is a 16-bit value For BLTZ, BGEZ the instruction after the branch (*delay slot*) is always executed.
BLTZAL BGEZAL	rA, destination rA, destination	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.)
J	target	pc <- pc(31:28) || target || 00 target is a 26-bit absolute. The instruction following J (delay slot) is always executed.
JAL	target	Same as above except that the address of the instruction following the delay slot is saved in r31.
JR	rA	pc <- (rA) The instruction following JR (delay slot) is always executed.
JALR	rA, rD	Same as above 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	The Sys Trap occurs if SYSCALL is executed.
BREAK	The Bp Trap occurs if BREAK is executed.
RFE	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(rA)	rCGEN <- Memory[rA + offset] Coprocessor z general reg rCGEN is loaded from data memory. The memory address is computed as *base + offset*, where the base is reg rA 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(rA)	rCGEN <- Memory[rA + offset] Coprocessor z general reg rCGEN is stored to data memory. The memory address is computed as *base + offset*, where the base is reg rA and the offset is the16-bit offset sign-extended to 32 bits.
MTCz rB, rCGEN CTCz rB, rCCON	In MTCz(CTCz), the general register rB is moved to Coprocessor z general (control) reg rCGEN(rCCON). rCGEN and rCCON cannot be referenced in the following instruction.
MFCz rB, rCGEN CFCz rB, rCCON	In MFCz(CFCz), the Coprocessor z general (control) reg rCGEN(rCCON) is moved to the general register rB. rB cannot be referenced in the following instruction.
BCzT destination BCzF destination	pc <- (pc + 4) + 14'(dest(15)) || dest || 00 if COND 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)

3.2.1. CEI Operations

Customers may add proprietary or application-specific opcodes to their LX8000 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, rA, immed	rD <- rA NEWOPI immed Reg rA 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
NEWOPR rD, rA, rB	rD <- rA NEWOPR rB Reg rA is supplied to the SRC1 port of CEI and reg rB 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 CEI decodes NEWOPI or NEWOPR, it must signal the Core that a custom operation has been executed so that the Reserved Instruction trap will not be taken. Multi-cycle custom operations may be executed by asserting CESEL.

Note: The custom operation may choose to ignore the SRC1 and SRC2 operands supplied by the CEI and reference customer registers instead. Results can also be written to an implicit customer register; however, unless D = 0 is coded, a register in the Core will also be written.

3.2.2. Interface Signals

Table 13: Custom Engine Interface Signals

Signal	Type (relative to core)	Description
SRC1[31:0]	OUTPUT	Operand supplied to customer logic.
SRC2[31:0]	OUTPUT	Operand supplied to customer logic.
RES[31:0]	INPUT	Result of customer logic. Supplied to Core.
CEIOP[11:0]	OUTPUT	Instruction OP and SUBOP fields – to be decoded by customer logic.
CEHALT	INPUT	Indicates that a multi-cycle custom operation is in progress.
CESEL	INPUT	Indicates that a CEI operation has been decoded.

3.3. Memory Management

The LX8000 includes a Simplified Memory Management Unit (SMMU) for the instruction memory address and the data memory address. These units are physically located in the Local Memory Interface (LMI) modules. The hardwired virtual-to-physical address mapping performed by the SMMU is sufficient to ensure execution of third-party software development tools.

Table 14: SMMU Address Mapping

Virtual Address Space	Description	Mapped to Physical Address
0xFF00_0000 to 0xFFFF_FFFF	EJTAG address space. 16 Mbyte. Uncached. This address range is reserved for EJTAG use only.	0xFF00_0000 to 0xFFFF_FFFF
0xC000_0000 to 0xFEFF_FFFF	KSEG2. 1Gbyte (minus 16 Mbyte). Addressable only in kernel mode. Cached.	0xC000_0000 to 0xFEFF_FFFF
0xA000_0000 to 0xBFFF_FFFF	KSEG1. 0.5 Gbyte. Addressable only in kernel mode. Uncached. Used for I/O devices.	0x0000_0000 to 0x1FFF_FFFF
0x8000_0000 to 0x9FFF_FFFF	KSEG0. 0.5 Gbyte. Addressable only in kernel mode. Cached.	0x0000_0000 to 0x1FFF_FFFF (differentiated from KSEG1 addresses with an internal signal)
0x0000_0000 to 0x7FFF_FFFF	KUSEG. 2Gbyte. Addressable in kernel or user mode. Cached.	0x4000_0000 to 0xBFFF_FFFF

Note: The 0.5 Gbyte of physical address space from 0x2000_0000 to 0x3FFF_FFFF is not accessible with the above memory map.

3.4. Exception Processing

The LX8000 implements the MIPS R3000 exception processing model as described below. Features specific to on-chip TLB support are not included. In the discussion below, the term *exception* refers to both *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. Note that Sys, Bp, RI, CpU can share the same priority level because only one can occur in a particular time slot.

Table 15: 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.
Ov	3	12	Arithmetic overflow trap. Can occur as a result of signed add or subtract operations.
Sys	4	8	SYSCALL instruction trap. Occurs when SYSCALL instruction is executed.
Bp	4	9	BREAK instruction trap. Occurs when BREAK instruction is executed.
RI	4	10	Reserved instruction trap. Occurs when a reserved opcode is fetched. Reserved opcodes are listed below.
CpU	4	11	Coprocessor Usability trap. Occurs when an attempt is made to execute a Coprocessor n operation and Coprocessor n is not enabled.
AdEL – data	5	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	6	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.
Int	7	0	Unmasked interrupt. There are six (6) level-sensitive hardware interrupt request signals into the LX8000 Core. Each is synchronized by the Core to the LX8000 system clock. In addition, program writes to CAUSE[9:8] are software-initiated interrupt requests. Each of the eight (8) requests has an associated mask bit in STATUS. Int is generated by any unmasked request (when Interrupts are globally enabled).

3.4.1. Exception Processing Registers: STATUS, CAUSE, EPC, BadVAddr

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

CU — CU[n] = 1(0) indicates that Coprocessor n is usable(unusable) in Coprocessor instructions.

BEV — Bootstrap Exception Vector. Selects between two trap vectors. (see below)

IM — Interrupt masks for the six hardware interrupts and two software interrupts.

KU/IE — KU = 0(1) indicates kernel (user) mode. In the LX8000, 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).
KUo | IEo | KUp | IEp | KUc | IEc forms 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 below)

STATUS — is read or written using MTC0 and MTF0 operations. On reset, BEV = 1, KUc = IEc = 0. The other bits in STATUS are undefined. The 0 fields are ignored on write and are 0 on read. It is recommended that the user explicitly write them to 0 to insure compatibility with future versions of the LX8000.

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

BD — Branch Delay. Indicates that the exception was taken in a branch or jump delay slot.

CE — Coprocessor Exception. In the case of a Coprocessor Usability exception, indicates the number of the responsible Coprocessor.

IP — Interrupt Pending. Each bit in IP(7:0) indicated an associated unmasked interrupt request.

ExcCode — The ExcCode listed above for the different exceptions are stored here when as exception occurs.

CAUSE — is read or written using MTC0 and MTF0 operations. The only program writable bits in CAUSE are IP(1:0), which are called *software interrupts*. CAUSE is undefined at reset. The 0 fields are ignored on write and are 0 on read.

EPC: Coprocessor 0 General Register Address = 14

EPC is a 32-bit read-only register which 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 will hold the address of the branch instruction and BD will be set in CAUSE. The branch will typically be re-executed following the exception handler.

BADVADDR: Coprocessor 0 General Register Address = 8

BADVADDR is a 32-bit read-only register containing the virtual address (instruction or data) which generated an AdEL or AdES exception error.

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 ➡(push)

KUp | IEp | KUc | IEc | 0 | 0

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 context 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 ➡(pop)

KUp | IEp | KUp | IEp | KUc | IEc

(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    EPC, r26    // r26 is a temporary storage register in the RALU
. . .
JR      r26
RFE
```

3.5. The Coprocessor Interface (CI)

Designers may implement up to three Coprocessors to interface with the LX8000. The contents of these Coprocessors may include up to thirty-two (32) 32-bit *general registers* and up to thirty-two (32) 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.

Lexra supplies a simple Coprocessor Interface (CI) model allowing the customer to easily interface a Coprocessor to the LX8000. The CI supplies a set of control, address, and data busses that may be tied directly to the Coprocessor general and special registers.

The CI is described in more detail in Section 7, LX8000 Coprocessor Interface.

4. LX8000 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 are expected to be executed in User mode and 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.

Table 16: Context Switching Instructions

Instruction	Syntax and Description
My Context	MYCX rD 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.
Post Event to Context	POSTCX rS, rT 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. If a MFCXC instruction is executed as the first instruction immediately following the POSTCX, it is unpredictable whether the new or old value of CXSTATUS is returned.

Instruction	Syntax and Description
Context Switch Unconditional	CSW rS Bits 15:8 (the WAIT-EVENT bits) from this 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.
Load Word Uncached with Context Switch	LW.CSW rT, displacement(base) The *displacement*, 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 *displacement* is sign extended and added to the contents of *base* 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. 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.
Load TwinWord Uncached with Context Switch	LT.CSW rT, displacement(base) The *displacement*, 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 *displacement* is sign extended and added to the contents of the register *base* 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. 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.
Load QuadWord Uncached with Context Switch	LQ.CSW rT, displacement(base) The *displacement*, 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 *displacement* is sign extended and added to the contents of the register *base* 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. 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
Load TwinWord	LTW rT, displacement(base) The *displacement*, 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 *displacement* is sign extended and added to the contents of the register *base* 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.
Write Descriptor	WD[.CSW] rS, rT, deviceID 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 *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. If the optional.CSW extension is specified, the processor performs a context switch after the execution of this instruction's delay slot.
Write Descriptor with Load Word Uncached and Context Switch	WDLW.CSW rD, rS, rT, deviceID 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 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.
Write Descriptor with Load Twinword Uncached and Context Switch	WDLT.CSW rD, rS, rT, deviceID 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. 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
Write Descriptor with Load Quadword Uncached and Context Switch	WDLQ.CSW rD, rS, rT, deviceID 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 quadword response. The processor performs a context switch after the execution of this instruction's delay slot. 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

Note: The following restrictions apply to the delay slot of any context switching instruction (CSW, LW.CSW, LT.CSW, LQ.CSW, WD.CSW, WDLW.CSW, WDLT.CSW and WDLQ.CSW):

All: No branch or jump type instruction. No MTCXC instruction.
[WD]LW.CSW, [WD]LT.CSW [WD]LQ.CSW: no 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 the figure, 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.

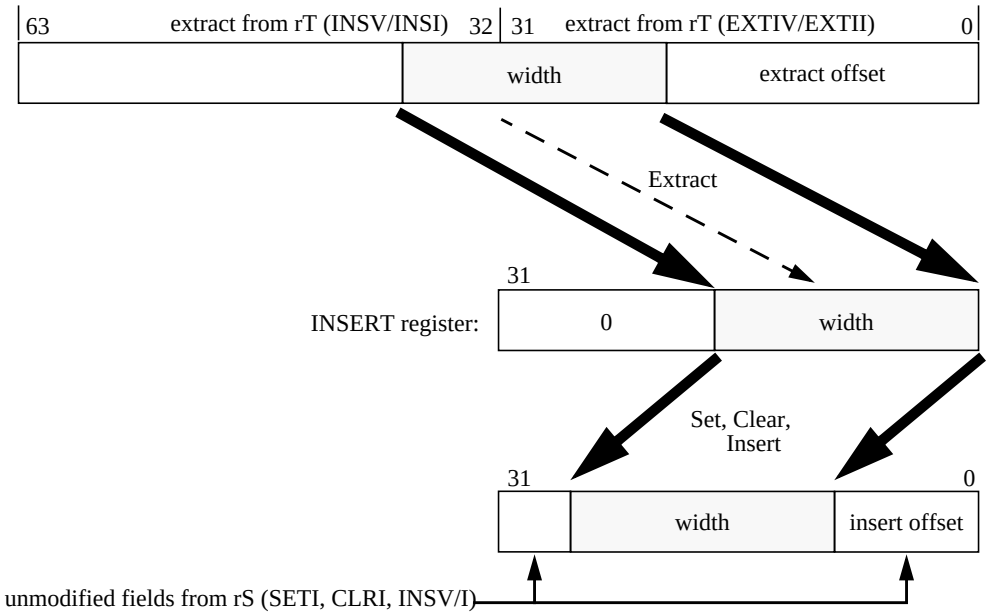


Figure 4: Insert and Extract Operations (Straddle Case)

Table 17: Bit Field Processing Instructions

Instruction	Syntax and Description
Set Bits Immediate	SETI rT, rS, width, offset 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.
Clear Bits Immediate	CLRI rT, rS, width, offset 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
Extract Bits for Insertion Variable	EXTIV rD, rS, rT 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.
Insert Bits Variable	INSV rD, rS, rT 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
Extract Bits for Insertion Immediate	EXTI<u>I</u> rD, <u>rT, width, offset</u> <u>The *offset* is a value n in the range 0-31. The *width* is a value m</u> 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.
Insert Bits <u>Immediate</u>	INS<u>I</u> rD, rS, rT, <u>offset</u> This instruction must be coded as the next sequential instruction in the program sequence after an EXTI<u>I</u>. 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 EXTI<u>I. The *offset* is a value p in the range 0-31.</u> 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.
Hash to Key	HASH rD, rS, 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 exact composition of the XOR subsets for each keysize is indicated in Table 18, Hash Instruction Key Bit Definition.

Instruction	Syntax and Description
Most Significant Bit Encode	MSB rD, rS, rT 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.
Jump to Offset Register	JOR rS, rT 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 runtime, 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.

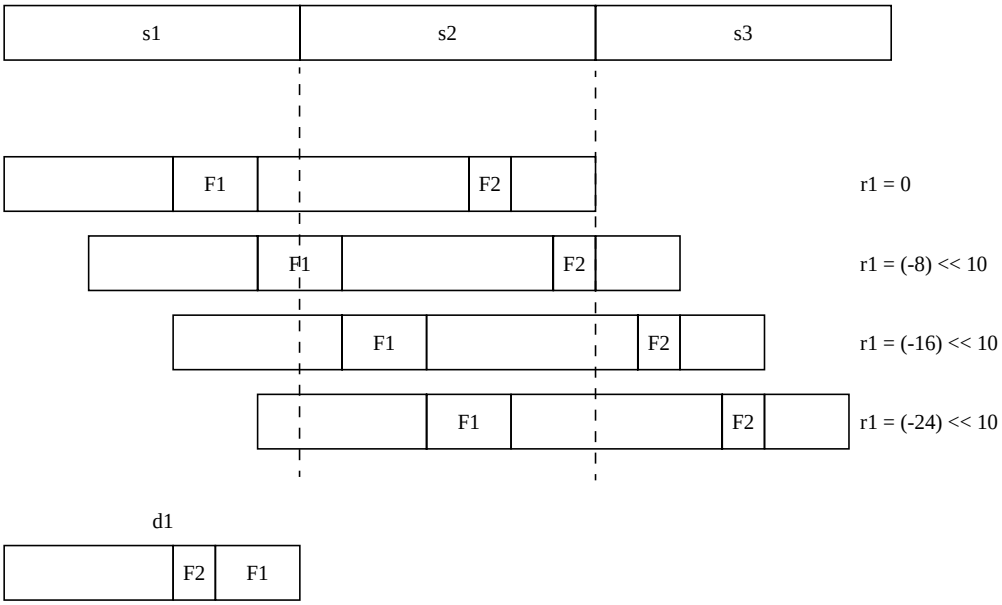


Figure 5: Packet Field Compaction with Variable Alignment

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
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

Keysize	KeyBit	Source Bits
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
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
Move From Context General Register	MFCXG rD, gT Bits MOVECX[2:0] are used to determine the target context *cntx*. The contents of general register gT in context *cntx* are loaded into the current context's general register rD
Move To Context General Register	MTCXG rT, gD 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.

Instruction	Syntax and Description
Move From Context Control Register	MFCXC rD, cT Bits MOVECX[2:0] are used to determine the target context *cntx*. The contents of control register cT in context *cntx* are loaded into the current context's general register rD.
Move To Context Control Register	MTCXC rT, cD 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
Dual Add for Checksum	ACS2 rD, rS, rT 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. LX8000 Instruction Summary and Encoding

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, deviceID	Write Descriptor to Device [with Context Switch]
WDLW.CSW	rD, rS, rT, dev	Write Descriptor to Device and Load Word/Twinword/ Quadword Uncached with Context Switch
WDLT.CSW	rD, rS, rT, dev	
WDLQ.CSW	rD, rS, rT, dev	
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

Instruction		Description
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

4.5.1. LX8000 Instruction Formats

The Lexra Formats are introduced into the MIPS instruction set by designating a single I-Format as "LEXOP2", then using the INST[5:0] "subop" field to permit up to 64 new Lexra opcodes. Thus the new opcodes model the MIPS "special" opcodes encoded in R-Format. The diagrams below illustrate the LEXOP2 codes using I-Format 011_110 which is unused in the MIPS I-V ISA.

[The default object code for LEXOP2 is 011_110. However, the location can be changed using a static reconfiguration. This helps insure compatibility of the extensions with future ISA extensions released by MIPS Technologies, Inc.]

This section also provides detail on the MFLXC0/MTLXC0 instructions which are variants of the MIPS standard MFC0/MTC0 instructions. These variants provide access to a space of Lexra defined Coprocessor 0 registers.

4.5.2. Load Formats

	31 26	25 21	20 16	15 6	5 0
Assembler Mnemonic	**LEXOP2 011 110**	**base**	**rt**	**immediate**	**Lexra SUBOP**
LW.CSW	LEXOP2	base	rt	displacement/4	LWC
LT.CSW	LEXOP2	base	rt-even, 0	displacement/8	LTC
LTW	LEXOP2	base	rt-even, 0	displacement/8	LTW
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.

4.5.3. Write Descriptor 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.

4.5.4. 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.

4.5.5. 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

4.5.6. Lexra-Coprocessor0 Register Access Instructions

	31 26	25 21	20 16	15 11	10 0
Assembler Mnemonic	**COP0 010 000**	**Copz rs**	**rt**	**rd**	**0**
MFLXC0	COP0	MFLX 00011	rt	rd	000 0000 0000
MTLXC0	COP0	MTLX 00111	rt	rd	000 0000 0000
	6	5	5	5	11

These are *not* LEXOP2 instructions. They are variants of the standard MTC0 and MFC0 instructions that allow access to the Lexra Coprocessor 0 registers listed below. As with any COP0 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

4.5.7. Lexra SUBOP Bit Encodings

Table 22: Lexra SUBOP Bit Encoding

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			

5. LX8000 Local Memory

5.1. Local Memory Overview

This chapter describes how memories are configured and connected to the LX8000 using the Local Memory Interfaces (LMIs). This section provides a brief summary of the conventions and supported memories. Section 5.2 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 chapter also discusses configuration options and the ports that customers must access to connect application specific RAM and ROM devices that are used by the LX8000 LMIs. All of the signals between the processor core, the LMIs, RAMs and the system bus controller are automatically configured by *lconfig*, the LX8000 configuration tool. *Lconfig* also produces documentation of the exact RAMs required for the chosen configuration settings, and writes RAM models used for RTL simulation.

The LMIs provide connection points for connect to RAMs that service the LX8000 processor's local instruction and data busses. The LMIs also provide the pathways from the processor to the system bus. The LX8000 includes an LMI for each of the local memory types. The sizes of the RAMs and ROMs are customer selectable. The LX8000 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. ROMs may also be connected, but may require a customer supplied address register at the address inputs.

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 LX8000 is configurable for 16, 32, 64, or 128 byte cache line size. The valid bits in all cache lines are automatically cleared by the LMIs upon reset. Data caches implement a write-through protocol. Caches do not snoop the system bus. The LX8000 is configurable to work with RAMs with a write granularity of 8 bits (byte) or 32 bits (word). Byte write granularity results in more efficient operation of store byte and store half-word instructions.

Table 23 summarizes the LMIs that can be integrated on the local busses.

Table 23: Local Memory Interface Modules

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

5.2. Cache Control Register: CCTL

CCTL. CP0 General Register Address = 20

31-8	7	6	5	4	3-2	1	0
Reserved	IROMOff	IROMOn	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.

Changes in the contents of the CCTL register are observed in the W stage. However, these changes affect instruction fetches currently in progress in the I stage, and data load or store operations in progress in the M stage.

The IROMOn and IROMOff bits of the CCTL register control the and use of the optional local IROM memory configured into the LX8000. When IROM is present and the LX8000 is reset, the LMI enables access to the IROM. A transition from 0 to 1 on IROMOff disables the IROM, allowing instruction references to be serviced IMEM, ICACHE or the system bus. A transition from 0 to 1 on IROMOn enables the IROM.

The IMEMFill and IMEMOff bits of the CCTL register control the contents and use of any local IMEM memory configured into the LX8000. When the LX8000 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 5.4. 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.

The ILock field controls set locking in the two set associative instruction cache. When ILock is 00 or 01, the instruction cache operates normally. When ILock is 10, all cached instruction references are forced to occupy set 1. The hardware will invalidate lines in set 0 if necessary to accomplish this. When ILock is 11, lines in set 1 are never displaced – i.e. they are locked in the cache. Set 0 is used to hold other lines as needed.

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 set 1, and use set 0 for other code.

The IInval and DInval fields control hardware invalidation of the instruction cache and data cache. A transition from 0 to 1 on IInval will initiate a hardware invalidation sequence of the entire instruction cache. Likewise, a 0 to 1 transition on DInval will initiate a hardware invalidation sequence of the entire data cache. The DMEM, if present, is unaffected by this operation.

The hardware invalidation sequence for the instruction and data caches requires one cycle per cache line to complete.

Depending on the circumstances, software may be able to employ an alternative to a full invalidation of the data cache. If a small number of lines must be invalidated, software may perform cached reads from aliases of

the memory locations of concern. This displaces data in the addressed locations of the data cache, even if they do not encache the affected memory location.

Another alternative, if the affected memory location has an alias in uncacheable (KSEG1) space, is to simply perform an uncached read of the affected memory locations. If the location is resident in the data cache it will be invalidated. This method has the advantage of not displacing data in the cache unless it is absolutely necessary to maintain coherency. Note that a write to a KSEG1 address has no affect on the contents of the data cache.

With either of these two alternatives, it is only necessary to reference one word of each affected cache line.

5.3. Instruction Cache (ICACHE) LMI

The ICACHE LMI supplies the interface for a direct mapped or two-way set associative instruction cache attached to the LX8000 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 set 0, and a second RAM for set 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 element of the set to replace, with element 0 being replaced if LRU is 0, and element 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 element should be returned to the CPU. In the following cycle, the ICACHE determines if the correct element 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	32 x 24 and 32 x 26 bits
2K bytes, 2-way	2 x 128 x 64 bits	64 x 23 and 64 x 25 bits
4K bytes, 2-way	2 x 256 x 64 bits	128 x 22 and 128 x 24 bits
8K bytes, 2-way	2 x 512 x 64 bits	256 x 21 and 256 x 23 bits
16K bytes, 2-way	2 x 1,024 x 64 bits	512 x 20 and 512 x 22 bits
32K bytes, 2-way	2 x 2,048 x 64 bits	1,024 x 19 and 1,024 x 21 bits
64K bytes, 2-way	2 x 4,096 x 64 bits	2,048 x 18 and 2,048 x 20 bits
1K bytes, direct mapped	128 x 64 bits	64 x 23 bits
2K bytes, direct mapped	256 x 64 bits	128 x22 bits
4K bytes, direct mapped	512 x 64 bits	256 x 21 bits
8K bytes, direct mapped	1,024 x 64 bits	512 x 20 bits
16K bytes, direct mapped	2,048 x 64 bits	1,024 x 19 bits
32K bytes, direct mapped	4,096 x 64 bits	2,048 x 18 bits

Configuration	ICACHE_INST RAM	ICACHE_TAG RAM
64K bytes, direct mapped	8,192 x 64 bits	4,096 x 17 bits

Table 25 lists the ICACHE signals that are connected to application specific modules. 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.
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 elements).
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.

5.4. 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 5.2Cache 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
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 modules. 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 address of the external resources which are accessed when an IMEM miss occurs. The *lconfig* utility supplied by Lexra will verify that the configured address range does not interfere with other regions defined for LX8000. 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.

Signal	Description
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.

5.5. Instruction ROM (IROM) LMI

The IROM LMI supplies the interface for an optional read-only local instruction store. The IROM serves a fixed range of the physical address space, determined by configuration settings in *lconfig*. IROM may be disabled via a hardware configuration pin, CFG_IROFF. IROM may also be enabled and disabled under software control as described in Section 5.2Cache Control Register: CCTL. The IROM is a convenient, low-cost alternative to a cache that makes read-only instruction memory available to the core for high-speed access.

The configurations supported by IROM, and the synchronous ROMs required for each, are summarized in Table 28.

Table 28: IROM Configurations

Configuration	IROM_DATA
no local instruction RAM	no ROM required
1K bytes, direct mapped	128 x 64 bits
2Kbytes, direct mapped	256 x 64 bits
4K bytes, direct mapped	512 x 64 bits
8K bytes, direct mapped	1,024 x 64 bits
16K bytes, direct mapped	2,048 x 64 bits
32K bytes, direct mapped	4,096 x 64 bits
64K bytes, direct mapped	8,192 x 64 bits
128K bytes, direct mapped	16,384 x 64 bits
256K bytes, direct mapped	32,768 x 64 bits

Table 29 lists the IROM signals that are connected to application specific modules. The IR_ prefix indicates signals that are driven by the IROM LMI module and received by the ROM. The IRR_ prefix indicates signals that are driven by ROM and received by the IROM LMI. The CFG_ prefix identifies configuration ports on the IROM LMI that are typically wired to constant values. Lexra supplies the Verilog module that makes all required connections to these wires. The width of the index and data lines depends upon the ROM connected to the LMI, and can be inferred from Table 27.

The CFG_ wires define where the IROM is mapped into the physical address space. This configuration information defines the local bus address region of the IROM. It also determines the address of the external resources which are accessed when an IROM miss occurs. The lconfig utility supplied by Lexra will verify that the configured address range does not interfere with other regions defined by the LX8000. Note that the size of the memory region must be a power of two, and must be naturally aligned.

Table 29: IROM ROM Interfaces

Signal	Description
IR_INSTINDEX	IROM index.
IRR_INSTRD	Instruction read data.
IR_INSTRE<N>	Instruction ROM read enable.
IR_INSTCS<N>	Instruction ROM chip select.
CFG_IRBASE[31:10]	Configured base address (modulo 1K bytes).
CFG_IRTOP[17:10]	Configured top address (bits that may differ from base).

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

5.6. Direct Mapped Write Through Data Cache (DCACHE) LMI

The DCACHE LMI supplies the interface for a direct mapped, write through data cache attached to the LX8000 local bus. The DCACHE LMI participates in cacheable data reads and writes, but only if the address is not claimed by the DMEM LMI. The configurations supported by DCACHE, and the synchronous RAMs required for each, are summarized in Table 30.

The direct mapped DCACHE module services word or twin-word read requests from the core in one cycle when the request hits the cache. Byte or half-word reads that hit the data cache require an extra cycle for alignment. The data cache can stream word and twin-word reads or writes that hit the cache at the rate of one per cycle. If the LX8000 is configured to work with RAMs that have word write granularity, byte or half-word writes that follow any write by one cycle and hit the cache require an extra cycle to merge the data with the current cache contents. Alternatively, the LX8000 can be configured to work with RAMs support byte write granularity, which eliminates the extra cycle.

Writes that are serviced by the data cache may require extra time to be serviced by the LBC if its write buffer is full. Also, when a cache write operation is immediately followed by a cache read, the cache must delay the read for one cycle while the write completes.

When a miss occurs, the cache obtains a cache line (4, 8, 16, or 32 words) of data from the Lexra Bus Controller (LBC). Write operations that hit the data cache are simultaneously written into the cache and forwarded to the write buffer of the LBC. Thus, if the core subsequently reads the data, it will likely be available from the cache. For main memory systems that support byte writes, all data writes that miss the cache are forwarded to the write buffer of the LBC, without disturbing any data currently in the cache. For main memory systems that can only write with word granularity, a byte or half-word write that misses the cache causes the cache to perform a line fill from main memory. The cache then merges the partial write data with the full word data obtained from memory, and writes the word to the system bus.

Table 30: DCACHE Configurations

Configuration	DCACHE_DATA RAM	DCACHE_TAG RAM
no data cache	no RAM required	no RAM required
1K bytes, direct mapped	128 x 64 bits	64 x 23 bits
2K bytes, direct mapped	256 x 64 bits	128 x 22 bits
4K bytes, direct mapped	512 x 64 bits	256 x 21 bits

Configuration	DCACHE_DATA RAM	DCACHE_TAG RAM
8K bytes, direct mapped	1,024 x 64 bits	512 x 20 bits
16K bytes, direct mapped	2,048 x 64 bits	1,024 x 19 bits
32K bytes, direct mapped	4,096 x 64 bits	2,048 x 18 bits
64K bytes, direct mapped	8,192 x 64 bits	4,096 x 17 bits

Table 31 lists the DCACHE signals that are connected to application specific modules. 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 30.

Table 31: 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.
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.

5.7. Scratch Pad Data Memory (DMEM) LMI

The DMEM LMI supplies the interface for a scratch pad data RAM attached to the LX8000 local bus. The DMEM module services in any cacheable or uncacheable data read or write operation that falls within its configured range.

Byte or half-word reads that hit the DMEM require an extra cycle for alignment. DMEM can stream word and twin-word reads or writes that hit DMEM at the rate of one per cycle. If the LX8000 is configured to work with RAMs that have word write granularity, byte or half-word writes that follow any write by one cycle and hit DMEM require an extra cycle to merge the data with the current DMEM contents. Alternatively, the LX8000 can be configured to work with RAMs support byte write granularity, which eliminates the extra cycle. Also, because a write operation to the DMEM is never sent to the LBC, writes to DMEM will not cause the LBC to stall the processor due to a full write buffer condition.

LX8000 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*, LX8000 does *not* tie the DMEM RAM into the RAM BIST paths or scan collar muxes. Other RAMs remain connected to these options.

The configurations supported by the DMEM, and the synchronous RAMs required for each, are summarized in the Table 32.

For LX8000, dual-port RAMs are used, and the second port is brought out the processor hierarchy for customer connection.

Table 32: DMEM Configurations

Configuration	DMEM_DATA RAM (64-bit)	DMEM_DATA RAM (128-bit)
no local data RAM	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 33 lists the DMEM signals that are connected to application specific modules. 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 32.

The *CFG_* wires define where the DMEM is mapped into the physical address space. This configuration information defines the local bus address region of the DMEM. 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 LX8000. The size of the memory region must be a power of two, and must be naturally aligned.

The DMEM LMI can also be used as a ROM controller simply by tying off the write enable and data input lines in the RAM wrapper, and instancing a ROM in the RAM wrapper

For the LX8000, the DMADW_* RAM ports are brought out of the lx2 hierarchy for customer connection.

Table 33: DMEM RAM Interfaces

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

Signal	Description
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.

6. LX8000 System Bus

6.1. Connecting the LX8000 to internal devices

The Lexra System Bus (LBus) is the connection between the LX8000 and other internal devices, such as system memory, USB, IEEE-1394 (Firewire), and an external bus interface. 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.

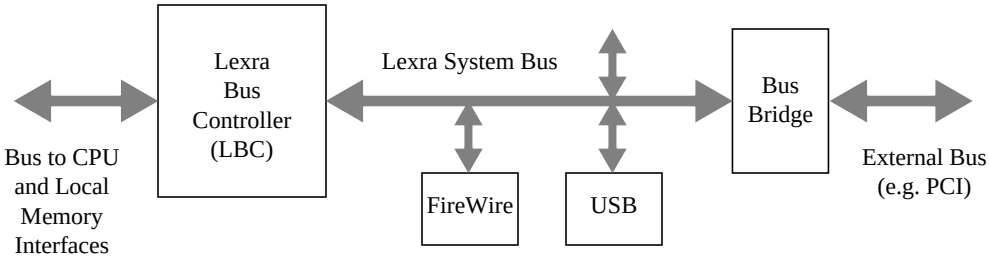


Figure 6: Lexra System Bus Diagram

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 for same-cycle turn-around. The LBC provides an optional asynchronous interface between the CPU and the Lexra bus, allowing the Lexra bus speed can be set to be any speed equal to or less than the CPU clock frequency.

The Lexra bus data path for the LX8000 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 words of data are transferred. The Lexra bus accomplishes this by transferring words of data from incremental addresses on successive clock cycles.

The LBC contains a write buffer. When the CPU issues a write request to a Lexra Bus device, the address and data are saved in the buffer and sent to the device sometime later. The CPU can continue processing, having safely assumed that the write will eventually happen. This is described more thoroughly in Section 6.8.2.

The LBC drives enabling signals to control muxes or tristate buffers. This allows the Lexra bus to have either a bi-directional or point-to-point topology.

6.2. 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, indicates as 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-cycle* 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.

Some signals on the Lexra bus are *active low*. That is, they are considered logically true when they are electrically low and logically false when electrically high. A device *asserts* a signal when it drives it to its logical true electrical state.

6.3. Bus Operations

The purpose of the Lexra bus is to connect together the various components of the system, including the LX8000 CPU, main system memory, I/O devices, and external bus bridges. Different devices have different transfer requirements. For example, the LX8000 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. The Lexra bus supports the various types of transfers needed by both I/O and the processor.

The nine types of bus operations are single-cycle read, line read, burst read, single-cycle write, line write (though this is not used by the LX8000 CPU), burst write, split read, write split read, and split data.

6.3.1. Single-Cycle Read

The single-cycle read operation reads a twinword, a 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.)

6.3.2. Read Line

The read line operation reads 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 LX8000 and the Lexra bus support a configurable line size, specified through *lconfig*. The default line size of four words (16 bytes) ia assumed here.

There are two ways that the target could transfer the data back to the initiator. The conventional way is to transfer four words of data in sequence, starting at the nearest 16-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.

Some memory devices may implement a performance optimization called *desired-word-first*. If the address is

not aligned to a 16-byte boundary, then the first data returned by the target is the word corresponding to the address instead of the first word of the line. The second word is the next sequential word of data and so on. At the end of the line, the target wraps around and returns the first word of line.

The LX8000 supports two ways of incrementing the address of a line refill. 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 the table below. The low-order address bits 3:2 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.

Table 34: Line Read Interleave Order

Interleaved	Address[3:2]			
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

6.3.3. 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 words 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 LX8000 will never issue a burst read operation.

Note that there is a difference between a 4-cycles 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.

6.3.4. Single-Cycle Write

The single-cycle write operation writes two words, a single word, a halfword, or a byte to the target.

The LX8000 uses a cache with a write-through policy. All CPU instructions that write to memory generate a single-cycle write operation. (Unless the address is in the local scratchpad memory, in which case the write operation will not make it out to the Lexra bus).

6.3.5. Line Write

The line write operation is not used by the LX8000. This operation could be used by a processor that has a data cache that implements a write-back policy.

6.3.6. 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 processor.

6.3.7. Split Read

The LBC issues a Split Read command when the processor executes an LW.CSW, LT.CSW or LQ.CSW instruction. For this command, the 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 command as a bus master, described below.

6.3.8. 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.

6.3.9. 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.

6.4. Signal Descriptions

Table 35: 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 a 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.

6.5. LBus Commands

The initiator drives BCMD during the cycle that BFRAME is asserted.

```
BCMD[8:6]    876
             000 read
             001 write
             010 split read
             011 write split read
             100 reserved
             101 split data
             110 reserved
             111 reserved
```

BCMD[5:4]	54[1]
	00 burst, fixed length[2]
	01 burst, unlimited number of words
	10 line, interleaved wrap[3]
	11 line, linear wrap
BCMD[3:0]	3210[4]
	1000 1 byte
	1001 2 bytes
	1010 3 bytes
	1011 1 word
	1100 2 words
	1101 reserved
	111x reserved
	0000 4 words
	0001 8 words
	0010 16 words
	0011 32 words
	01xx reserved

1. CMD[5:4] must be 00 for Split Read, Write Split Read and Split Data commands.
2. The number of words comes from BCMD[2:0]
3. Length is determined by the Line size, not BCMD[3:0]
4. CMD[3:0] indicates the read data size for split read, write split read and split data commands.

6.6. Byte Alignment

The Lexra Bus is a big endian bus. Transactions must have their data driven to the appropriate bus rails. The bus mapping is as shown in Table 36.

Table 36: LBus Byte Lane Assignment

		Lexra Bus data byte lanes used							
BCMD[1:0]	**ADDR[2:0]**	**63:56**	**55:48**	**47:40**	**39:32**	**31:24**	**23:16**	**15:8**	**7:0**
00	000	X							
00	001		X						
00	010			X					
00	011				X				
00	100					X			
00	101						X		
00	110							X	
00	111								X
01	000	X	X						
01	010			X	X				
01	100					X	X		
01	110							X	X
11	000	X	X	X	X				
11	100					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 word.

6.7. Split Transactions

There are several processor context switch instructions that can cause split transactions on the LBus. 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.

The LBC can issue a split read request with either a read or write transaction. When the processor executes a LW.CSW, LT.CSW or LQ.CSW instruction, a simple split read request (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] is used to indicate the size of the requested read data.

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 will accept this command

and receive 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. The GTID is made up of two values, the Processor Number and Context Number. The lower 4 bits of the GTID are used for the Context Number. The Processor Number is an 8-bit field. The use and interpretation of the Reserved field is processor dependent. A target system bus device must save all 16 bits of the GTID value driven with a split read request, and subsequently drive the 16-bit value onto GTID during the split read data transaction.

Table 37: LBus GTID Fields

GTID[15:12]	GTID[11:4]	GTID[3:0]
Reserved	ProcNum	ContextNum

The GTID is driven by an LBC for all transactions.

6.8. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the element of the LX8000 that connects to the Lexra Bus. It forwards all transaction requests from the LX8000 CPU to the Lexra Bus.

6.8.1. LBC Commands

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

Table 38: LBus Commands Issued by the LBC

Command	BCMD[5:4]	BCMD[3:0]	Circumstances
Read Line	10 or 11, depending on configuration	0000	A cache miss during a read by the CPU
Read Single (word/halfword/byte)	00	10xx	A read by the CPU from an address in uncachable address space
Write Single (twinword/word/half-word/byte)	00	1xxx	A write by the CPU into cacheable or uncachable address space
Split Read	00	1011, 1100 or 0000	An LW.CSW, LT.CSW or LQ.CSW instruction is executed
Write Split Read	00	1011,1100 or 0000	A WDLW.CSW, WDLT.CSW or WDLQ.CSW instruction is executed.

6.8.2. LBC Write Buffer and Out-of-Order Processing

The LBC contains a write buffer with a depth that is configurable with *lconfig*. All write requests from the CPU are posted in the write buffer. The CPU will not wait for the write to complete. Write operations complete in the order they are entered into the queue. If the queue fills, then the CPU must wait until an entry becomes available.

When the CPU issues a read operation, the LBC will attempt to forward that request to the Lexra Bus ahead of any pending write operations. This significantly improves performance since the CPU needs to wait for the read operation to complete and would waste time if it had to also wait for unnecessary or irrelevant writes to complete.

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

1. The address of a pending write is within the same cache line as the read. The LBC 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. The read is to uncacheable address space. All writes will 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 LBC will hold 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 so that reads will never pass writes.

6.8.3. 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 cache miss. When the LBC is configured with a synchronous interface, the cache can accept the data as fast as the LBC can read it. Therefore, there is no need for a large read buffer. Customers may reduce the size of the read buffer to a minimum size of two 64-bit entries.

In some cases, there is a need to minimize the number of gates. The read buffer size may be reduced to two or four entries for the asynchronous case. This causes a penalty in terms of Lbus utilization since now the LBC may have to de-assert 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 relatively rare since simultaneous instruction cache and data cache misses are relatively rare. For a smaller read buffer, IRDY deassertion is almost a certainty.

6.8.4. Transfer Descriptions

This section describes the various types of read and write transfers in detail. These operations follow certain patterns and rules. The rules for driving and sampling the bus are as follows:

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 some 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 dead cycle. If an initiator gives up the bus, another initiator needs to wait for one dead 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 for the data bus to turn around.

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 later in this chapter).

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 word of data. In most cases, the initiator will asserts BIRDY to indicate that it is ready to receive data (or read operations) or is driving valid data (for write operations). If the operation is a write, the initiator will drive valid data onto BDATA.

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

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

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

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

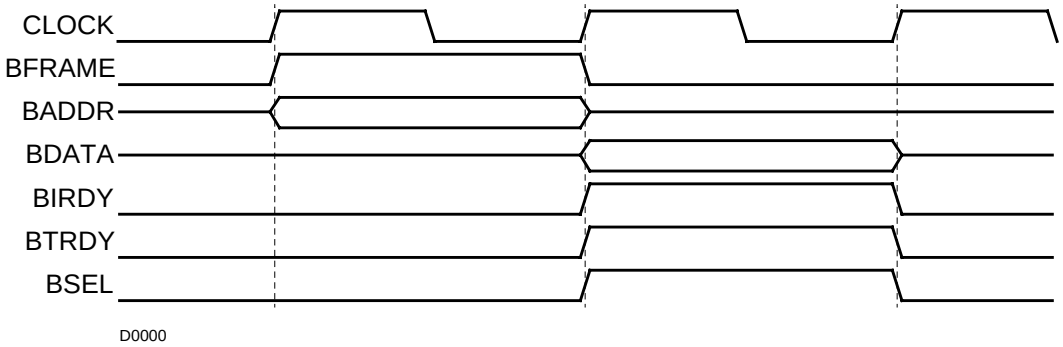
7. The arbiter grants the bus to the next initiator, and may do so during a bus transfer by a different initiator. The new initiator must sample BFRAME and BIRDY. When both BIRDY and BFRAME is sampled de-asserted and the new initiator has been given grant, it can assert BFRAME the next cycle to start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

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

6.8.5. Single Cycle Read with No Waits

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



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

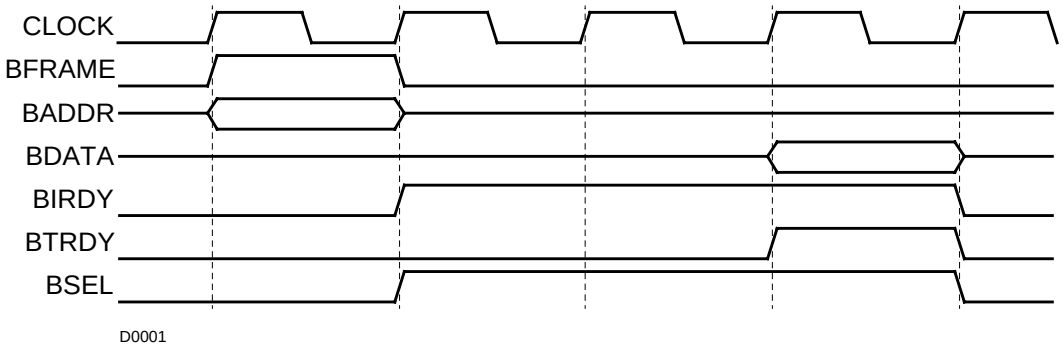
1. Initiator asserts BFRAME and drives BADDR.

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.

6.8.6. Single Cycle Read with Target Wait

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



This is a common single-cycle 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.

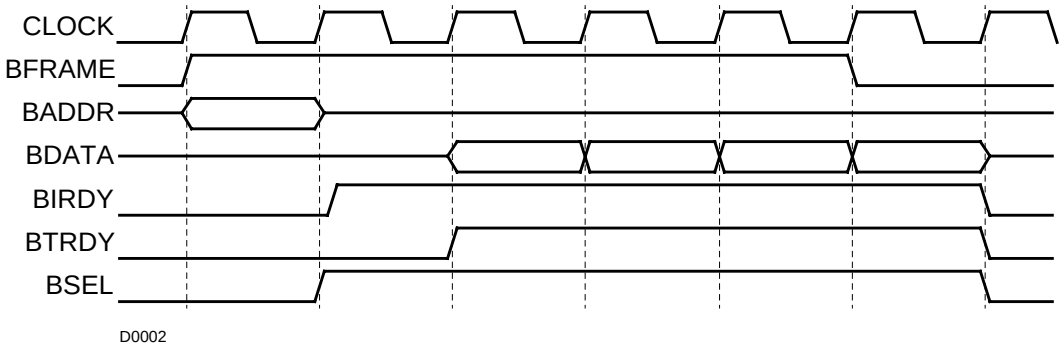
3. Target has not asserted BTRDY so no data is transferred.

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.

6.8.7. Line Read with No Waits

This operation is used to service a cache miss. Four twinwords of data are transferred in sequence. In this example, the target is supplying four twinwords of data without any waits.



1. Initiator drives BADDR and asserts BFRAME to indicate beginning of transaction.

2. Target asserts BSEL to indicate that it had decoded the address and will send data when it is ready. Initiator asserts BIRDY to indicate that it is ready to receive data.

3. Target drives data and asserts BTRDY.

4. Target drives second twinword of data and continues to assert BTRDY.

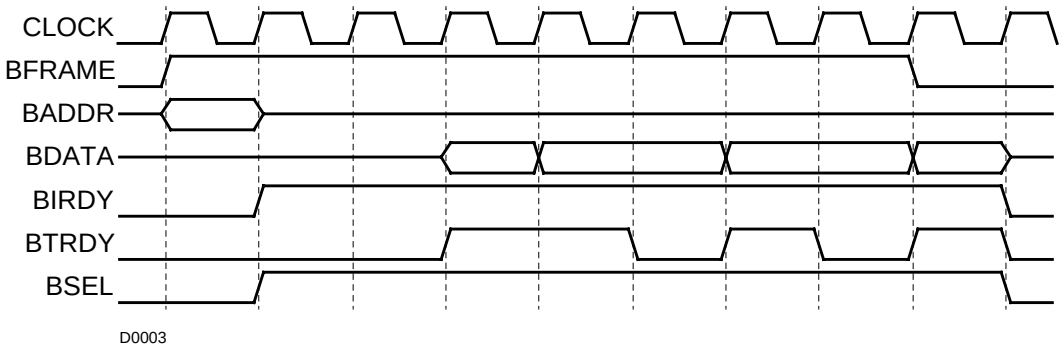
5. Target drives third twinword of data and continues to assert BTRDY.

6. Target drives last twinword of data. Initiator de-asserts BFRAME to indicate that the next word of data 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.

6.8.8. Line Read with Target Waits

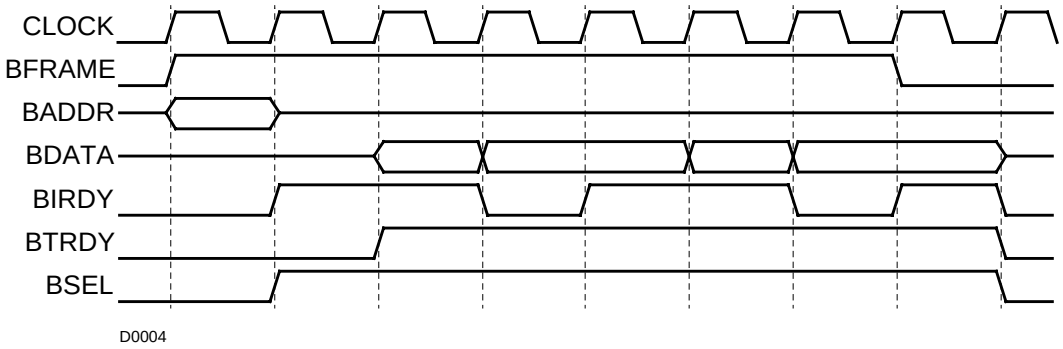
This illustrates what happens when a target needs extra time to fetch data it needs to service a cache miss.



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 twinword of data and asserts BTDRY.
5. Target drives second twinword of data and asserts BTRDY.
6. Target cannot get third twinword of data, so it de-asserts BTRDY.
7. Target drives third twinword of data and asserts BTRDY.
8. Target cannot get fourth twinword of data, so it de-asserts BTRDY.
9. Target drives fourth twinword of data and asserts BTRDY.

6.8.9. 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 twinword of data; initiator cannot accept it, so it de-asserts BIRDY.

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

6. Target drives third twinword of data; initiator accepts it.

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

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

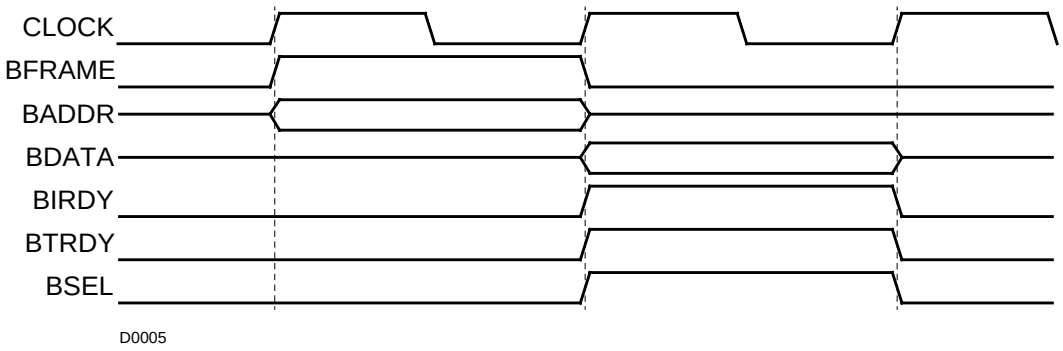
6.8.10. Burst Read

This is identical to the read line

6.8.11. Single-Cycle Write with No Waits

A single-cycle write operation occurs almost every time the LX8000 processor executes a store instruction. This is because the cache used in the processor uses a write-through policy. Of course, writes to uncacheable address space and to an I/O device will also generate a single-word write. Single-word write operations are used to write words, halfwords and bytes.

A single-word write without waits requires two cycles.



1. Initiator asserts BFRAME and drives address.

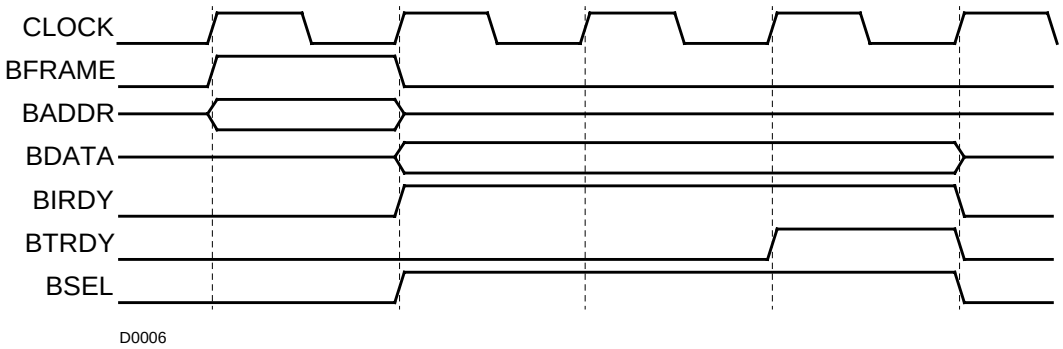
2. Target samples address and asserts BSEL. Initiator drives data and asserts BIRDY. In this case, target is also able to accept data, so it asserts BTRDY. Initiator also de-asserts BFRAME to indicate that it is ready to send the last (and only) word of data.

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

6.8.12. Single-Cycle Write with Waits

This is an example of a single-cycle write operation where the target cannot immediately accept data and

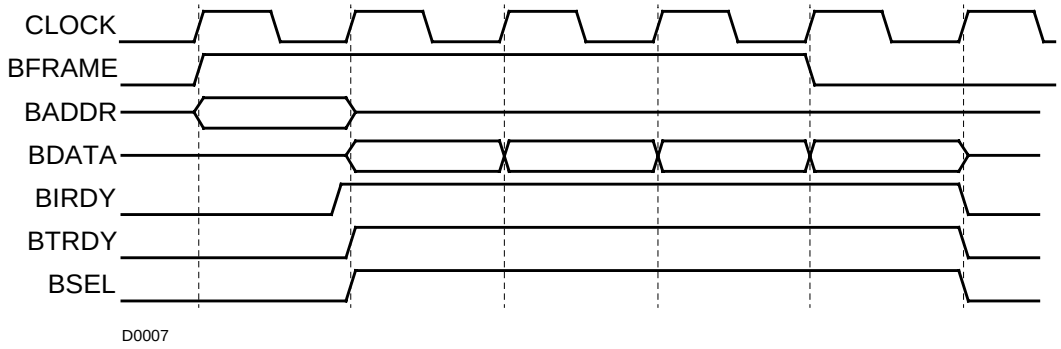
must insert wait states.



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.

6.8.13. Burst Write with No Waits

A burst write operation is generally used to transfer large amounts of data from an I/O device to memory via a DMA transfer. The following illustrates a best-case scenario with no wait states.

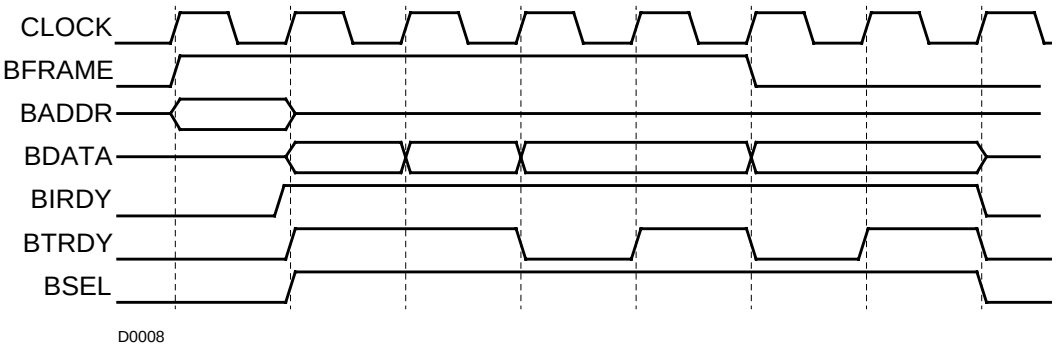


1. Initiator drives address and asserts BFRAME.
2. Target asserts BSEL and BTRDY to indicate it will accept data. Initiator drive data and asserts BIRDY.
3. Initiator drives next twinword of data; target continues to accept data and indicates as such by continuing to assert BTRDY.
4. Initiator drives third twinword of data; target continues to accept.
5. Initiator drives fourth twinword of data and de-asserts BFRAME to indicate that this will be its last word sent; target accepts data.
6. Target de-asserts BTRDY and BSEL; initiator gives up control of the bus by de-asserting BIRDY.

6.8.14. Burst Write with Target Waits

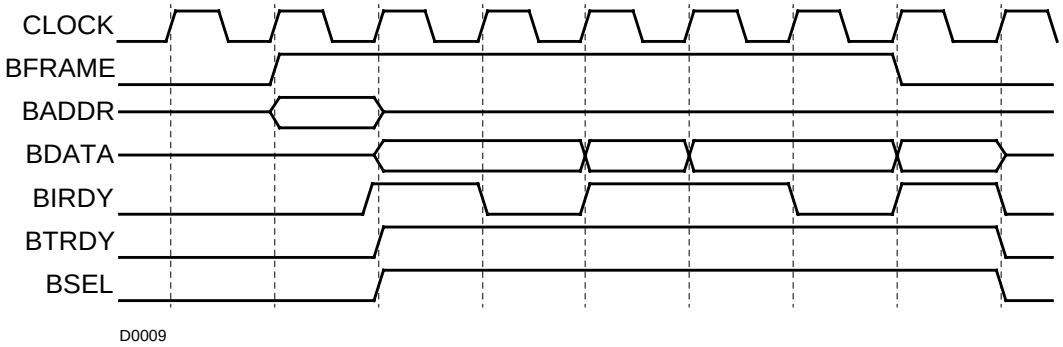
This example is similar to the above example, except that during the third and fourth data word transfer, the target cannot accept the data quickly enough, so it de-asserts BTRDY which indicates to the initiator that it

should hold the data for an additional cycle.



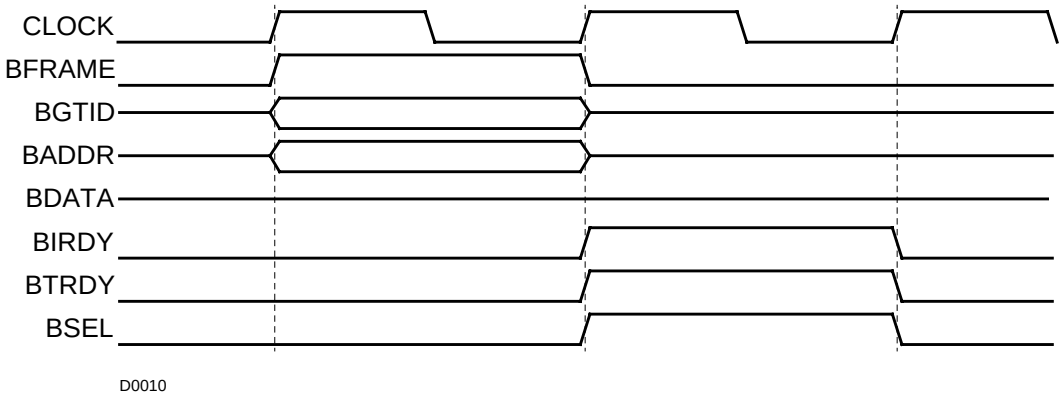
6.8.15. Burst Write with Initiator Waits

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



6.8.16. Split Read command

An LBC issues a Split Read command 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 deasserted and ADDR, CMD, GTID are not driven. IRDY is asserted.

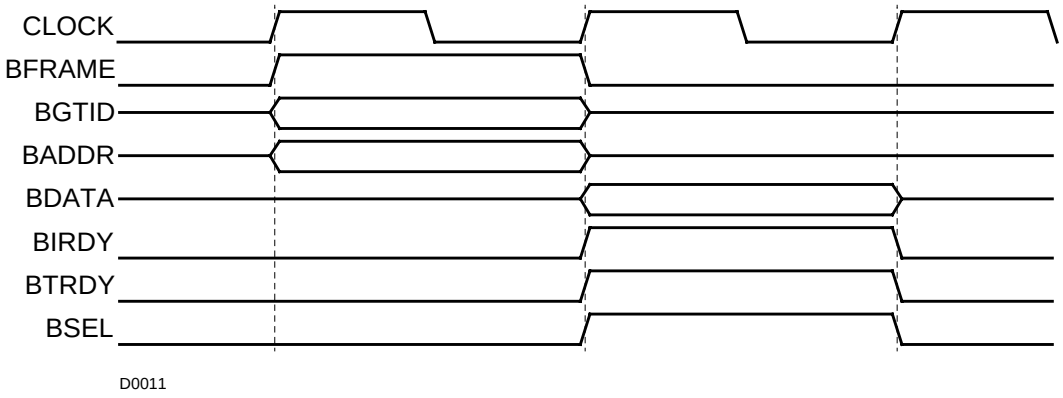
3. The LBC deasserts IRDY and the target device deasserts SEL and TRDY to indicate the split read request transaction is complete.

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.

6.8.17. 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 deasserts 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 deasserts IRDY and the target device deasserts 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.

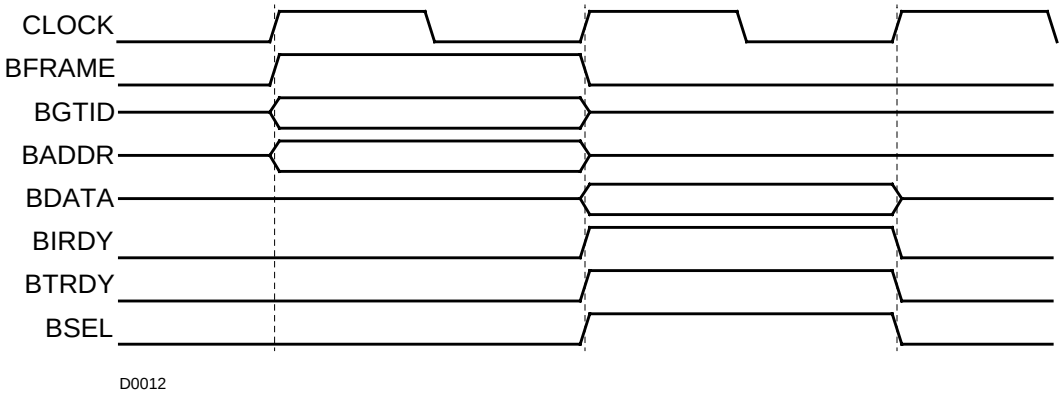
Since write data is being transferred with these transactions, data stalls are allowed. The rules for TRDY and

IRDY are the same for these write transactions as they are for regular write transactions. An LBC will never de-assert IRDY to cause a data stall during 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.

6.8.18. 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 and indicates it is two word read. 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 deasserted while IRDY is asserted. ADDR, CMD and GTID are only driven the first cycle.

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

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

The transaction can supply one or two words of data on the bus, as indicated by CMD[3:0]. For the one word case, the word is aligned on the data bus based on the original read address, according to the rules shown in Table 36 on page 75.

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

6.9. 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. All Split Reads are uncacheable and therefore are not allowed to bypass pending writes.

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.

6.10. LBC Signals

The table below summarizes the LX8000 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 39: 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	BRAME	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

I/O	LBC Port	Bus Signal	Description
output	LDoe[7:0]	-	LBC data output enable terms
output	LToe]	-	LBC transaction output enable terms

6.11. Arbitration

6.11.1. Rules

The following are the rules for arbitration (GNT=grant, REQ=request):

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 it is the last data phase of the previous transaction when master samples asserted GNT, master may assert FRAME on next cycle.

3. If the bus is busy when the master samples GNT, is must also snoop FRAME, IRDY and Trdy. One cycle after FRAME is not asserted and both IRDY and TRDY are asserted (indicating the last data phase), if GNT is still asserted, master may now drive FRAME (i.e. GNT & ~Frame_R & (Irdy_R & Trdy_R)).

6.11.2. LBC behavior

The LBC, when it need access to the bus, asserts REQ and in the same cycle samples GNT, ~FRAME, and either ~IRDY or (IRDY & TRDY). If these are true, then the LBC will on the next cycle take ownership of the bus. REQ is deasserted on the cycle after LBC asserts FRAME. If the bus is busy, LBC continues to snoop these four signals for this condition. All other Lbus arbitration rules can be based on this behavior of the LBC.

6.12.Connecting Devices to the Bus

There are three sets of output enables: TOE(valid for the length of the transaction), COE (valid for only the first cycle of a transaction), and DOE (valid for data transfers, asserted by the master for writes and by the slave for reads).

TOE is intended to qualify:

```
FRAME
IRDY
```

COE is intended to qualify:

```
CMD
ADDR
GTID
```

DOE is intended to qualify:

```
DATA
```

There is no output enable to qualify TRDY and SEL. These are defined by customer logic for slave devices.

Instead of using TOE it may be desirable to instead OR all of the FRAME signals, either centrally or one OR gate for each target and master. The same holds true for IRDY, TRDY, and SEL. 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.

Therefore, it is defined that masters and slaves not taking part in a transaction always keep FRAME, IRDY, TRDY, and SEL driven and de-asserted.

7. LX8000 Coprocessor Interface

The LX8000 processor provides customer access points for the Coprocessor Interfaces. This section provides a description of these access points. Attachment of memory devices to the LMIs, the System Bus, and the EJTAG interface are described in separate chapters.

7.1. 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 core's general registers with the MTCz instruction. Data is moved from the coprocessor's general registers to the core'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 core'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 core. The condition flag can be a bit of a control register or a logical function of several control register values. 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).

7.2. Coprocessor Interface (CI) Signals

The CI provides the mechanism to attach the custom coprocessor to the core. 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.

Table 40: Coprocessor Interface Signals

Signal	Direction	Description
C<z>condin	input	Cop branch flag.
C<z>rd_addr[4:0]	output	Cop read address.
C<z>rhold	output	Cop hold condition, one stalls coprocessor.
C<z>rd_gen	output	Cop general register read command.
C<z>rd_con	output	Cop control register read command.
C<z>rd_data[31:0]	input	Cop read data.
C<z>wr_addr[4:0]	output	Cop write address.
C<z>wr_gen	output	Cop general register write command.
C<z>wr_con	output	Cop control write address command.
C<z>wr_data[31:0]	output	Cop write data.
C<z>invld_M	output	Cop invalid instruction flag, one indicates invalid instruction in M stage.

Signal	Direction	Description
C<z>xcpn_M	output	Cop exception flag, one indicates exception in M stage.
C<z>rd_cntx[2:0]	output	Cop read context number.
C<z>wr_cntx[2:0]	output	Cop write context number.

The addresses, output data, and control signals are supplied to the user's Coprocessor on the rising edge of the system clock. In the case of a read cycle, the coprocessor must supply the data from either the control or general register on C<z>rd_data by the end of the same cycle. Similarly, the write of data from C<z>wr_data to the addressed control or general register must be complete by the end of the cycle.

The CI incorporates a forwarding path so that data which is written in Instruction(N) can be read in instruction (N + 2). The Coprocessor registers should be implemented as positive-edge flip-flops using the LX8000 system clock.

7.3. Coprocessor Write Operations

During a coprocessor write, the CI sends C<z>wr_addr and C<z>wr_data, and asserts either C<z>wr_gen or C<z>wr_con. The coprocessor must ensure that the coprocessor completes the write to the appropriate register on the subsequent rising edge of the clock. The target register is a decoding of C<z>wr_addr, C<z>wr_gen and C<z>wr_con. Use these instructions to cause a coprocessor write: LWCz, MTCz, and CTCz.

7.4. Coprocessor Read Operations

During a coprocessor read, the CI sends C<z>rd_addr and asserts either C<z>rd_gen or C<z>rd_con. The coprocessor must return valid data through C<z>rd_data in the following clock cycle. If the core asserts C<z>rhold, indicating that it is not ready to accept the coprocessor data, the coprocessor must hold the previous value of C<z>rd_data. The target register for the read is a decoding of C<z>rd_addr, C<z>rd_gen, and C<z>rd_con. The instructions causing a coprocessor read are SWCz, MFCz, and CFCz.

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 core'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 core 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 code does not access data from the coprocessor until N instructions after the coprocessor operation has stared. This is the least desirable method as it depends on the relative execution of the core and coprocessor. It can also complicate software debug.
- Have the coprocessor send an interrupt to the core, and the service routine for that interrupt accesses the appropriate coprocessor registers.
- Have the coprocessor set the C<z>condin 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 C<z>condin bit off
nop                     #branch delay slot
mfc2        $4, $7      #get results from COP2 general register $7
```

7.5. Coprocessor Interface and Pipeline Stages

Coprocessor writes occur in the W stage of the instruction pipeline. For coprocessor reads, the core generates address, rd_gen, and rd_con signals during the S stage, and the coprocessor returns data during the E stage which is passed by the CI to the core in the M stage. The core introduces a pipeline bubble after coprocessor instructions to ensure that the result of a MTCz instruction can be used by the immediately following instruction.

In particular, if there are back-to-back MTCz and MFCz instructions that access the same coprocessor register, the pipeline bubble still does not allow a cycle between the W stage write and E stage read as required. In this case a special forwarding path within the CI is used. That is, the “true” data from the coprocessor is ignored. Instead the exact data from the MTCz is used.

```
mtc2      I D S E M W
bubble      I D . . . .
mfc2          I D S E M W  # data forwarded by CI from mtc2
  wr_gen (W)        X
  rd_gen (S)      X
  rd_data(E)        X
```

The forwarding path can cause side effects if the coprocessor does not implement all of the bits of a register, contains read-only bits, or updates the register value upon reading the register. In such cases, the mfc2 instruction returns different data from what it would if the core did not activate the forwarding path. To avoid the forwarding path, another instruction must be inserted between the mtc2 and mfc2:

```
mtc2   I D S E M W
bubble   I D . . . .
foo        I D S E M W
mfc2         I D S E M W  # read data from coprocessor
  wr_gen (W)     X
  rd_data(E)       X
```

7.5.1. Pipeline Holds

The coprocessor must register the read address and the control signals rd_gen and rd_con. It must hold the (E stage) registered values of these signals when C<z>_rhold is active high, and should make the read data output a function of the (E stage) registered read address and control signals.

The wr_addr, wr_data, wr_gen and wr_con signals need not be registered. The coprocessor may decode these (W stage) signals directly to the appropriate register.

7.5.2. Pipeline Invalidation

Under certain circumstances the instruction pipeline can contain an instruction that must be discarded. This

can 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 wr_gen and wr_con for valid instructions. The coprocessor does not need to qualify these controls.

For the coprocessor read-type instructions, the CI may issue the S stage control signals rd_gen and rd_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 C<z>_xcpn_m and C<z>_invld_m as follows:

The signal C<z>_xcpn_m signal is used to discard any S stage (from CI) rd_gen or rd_con signal and any E stage (registered in the coprocessor) rd_gen or rd_con signal. It indicates that a preceding instruction in the pipe has taken an exception and that subsequent instructions in the pipe must be discarded.

The signal C<z>_invld_m signal is used to invalidate the operation of the current instruction in the M stage. This can be for various reasons not limited to an exception on a preceding instruction. If the coprocessor cannot tolerate speculative reads, it must register an M stage version of rd_gen and rd_con. The coprocessor must use the C<z>_rhold signal to hold this M stage version (as well as the E stage version). If C<z>_invld_m is asserted, then any such M stage signals must be discarded. To summarize, a rd_gen or rd_con instruction can "retire" only if it reaches the M stage and neither C<z>_rhold nor C<z>_invld_m is asserted.

8. LX8000 EJTAG

8.1. Introduction

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-effect, I/O constrained manner.

Lexra's EJTAG solution meets all such requirements. It uses existing IEEE JTAG pins as well as fast bring-up on new designs. It provides a way 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 Lexra Bus, bypassing internal caches and memories.

Software debug is enhanced by EJTAG features that allow 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 trace capabilities from which an accurate program execution history is derived. For the code-system perspective, PC profiling provides statistical analysis of code usage to aim code optimization.

8.2. Overview

A debug host computer communicates to the EJTAG probe through either a serial or parallel port or Ethernet connection. The probe, in turn, communicates to the LX8000 EJTAG hardware via the included IEEE 1149.1 JTAG interface. Through the use of the JTAG TAP controller, probe data is shifted into to the EJTAG data and control registers in the LX8000 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 cache address, data cache address or data cache data values.

When EJTAG is included in a configuration 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 LX8000. Information on these products is available at the following web sites.

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

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

LX8000 EJTAG implements all required features of version 2.0.0 of the EJTAG specification, and includes support for the following features:

- Processor access of host via addressing of probe memory space.
- Host probe can DMA directly into system memory or I/O devices.
- Hardware breakpoints on internal instruction and data busses.
- Single-step execution mode.
- Real-time Program Counter Trace.
- Debug exception and two new debug instructions: one for raising a debug exception via software, and one for returning from a debug exception.

8.2.1. 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 41: EJTAG Pinout

Signal Name	Direction	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 42: 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 43: 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

8.3. Single Processor PC Trace

The LX8000 EJTAG includes support for real-time Program Counter Trace (PC Trace). When in PC Trace

1. Based on EPI Interface Specifications for MAJIC^{TM} and MAJIC$^{PLUS\ TM}$
2. Based on 25ns JTAG clock period.

mode, the LX8000 will serially output a new value of the program counter whenever a change in program control occurs (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 LX8000: 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 LX8000's 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.

8.3.1. 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.

8.3.2. PC Trace PCST - Program Counter Status Trace

The Program Counter Status (PCST) output comprises N sets of 3-bit PCST values, where N is configurable as 1, 2, 3 or 4 via *lconfig*. A PCST value is generated every SYSCLK cycle. When DCLK is slower than the LX8000'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

8.3.3. 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 sent out of TPC. 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.

The TPC bus also outputs the exception type when an exception occurs. The exception type field-width is either 3- or 4-bits depending on whether or not vectored interrupts are present. This is covered in more detail below.

To reduce pinout, the TDO output is used for the least significant bit of TPC (or the only bit if "M" is set to 1).

8.3.4. Single-Processor PC Trace Pinout

Table 44: 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 45: 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

8.3.5. Vectored Interrupts and PC Trace

The EJTAG PC Trace facility specifies a 3-bit code be output on the TPC output when an exception occurs (the PCST pins give the EXP code). In order to distinguish the eight vectored interrupts in the LX8000 from all other exceptions, a 4-bit code is used instead.

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

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

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

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

For probes that do not support a 4-bit exception code, the LX8000 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.

8.3.6. Demultiplexing of TDO and TDI During PC Trace

In normal EJTAG PC Trace, TDI and TDO are multiplexed with the debug interrupt (DINT) and the lsb of the TPC (TPC[0]) when in PC Trace mode. This reduces the number of pins required by PC Trace, but has the unfortunate side-affect of preventing any access to EJTAG registers during PC Trace.

In order 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 demultiplexed such that TRST is used as TPC[0] and DINT is generated via EJTAG registers. Note: setting this option may require changes in EJTAG probe hardware. Check with probe manufacturer for details.

8.4. Data Break Exceptions for LX8000

The existing EJTAG data match architecture does not allow matches for some of the transaction types in the LX8000. This is described in more detail below.

8.4.1. 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).

8.4.2. Data Breaks on Write Descriptor Accesses

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

8.4.3. 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. LX8000 Lconfig Forms

A.1. Configuration Options for the LX8000 Packet 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.

```
PRODUCT            -- Lexra Processor name
PRODUCT_TYPE       -- indicates product type
TECHNOLOGY         -- identifies target technology
TESTBED_ENV        -- identifies simulation testbed environment type
RESET_TYPE         -- flip-flop reset method
RESET_DIST         -- reset distribution method
SLEEP              -- include clock SLEEP support
RESET_BUFFERS      -- reset buffers at top-level module
CLOCK_BUFFERS      -- clock buffers at top-level module
RAM_CLOCK_BUFFERS -- LMI RAM clock distribution method
COP1               -- coprocessor interface 1
COP2               -- coprocessor interface 2
COP3               -- coprocessor interface 3
CE0                -- custom engine 0
CE1                -- custom engine 1
M16_SUPPORT        -- 16-bit opcode support
MEM_LINE_ORDER     -- cache line fill beat ordering
MEM_FIRST_WORD     -- cache line fill first word
MEM_GRANULARITY    -- main memory system partial word write support
SYSTEM_INTERFACE   -- system bus interface type
WDESC_ADDR         -- Write Descriptor upper address bits
LBC_WBUF           -- Lexra Bus Controller write buffer depth
LBC_RBUF           -- Lexra Bus Controller read buffer depth
LBC_RDBYPASS       -- Lexra Bus Controller read bypass enable
LBC_SYNC_MODE      -- LBC synchronous/asynchronous selection
LINE_SIZE          -- cache line size, in words
ICACHE             -- instruction cache size
DCACHE             -- data cache size
IMEM               -- local instruction RAM with line valid bits
IROM               -- local instruction ROM
DMEM_WIDTH         -- local scratch pad data memory width
DMEM               -- local scratch pad data RAM
LMI_DATA_GRANULARITY -- DCACHE and DMEM write granularity
LMI_RANGE_SOURCE   -- source of LMI address ranges
LMI_RAM_ARB        -- allow external agents to arbitrate for LMI RAMs
JTAG               -- Internal JTAG Tap controller with EJTAG support
EJTAG              -- EJTAG Debug Support
EJTAG_INST_BREAK   -- Number of instruction breaks to be compiled
EJTAG_DATA_BREAK   -- Number of data breaks to be compiled
JTAG_TRST_IS_TPC   -- TRST pin is TPC out, instead of TDO/TPC mux
PC_TRACE           -- EJTAG PC trace pins
EJTAG_DCLK_N       -- EJTAG PCTrace DCLK N parameter
EJTAG_TPC_M        -- EJTAG PCTrace TPC M parameter
EJTAG_XV_BITS      -- EJTAG PCTrace number of Exception Vector bits
EJTAG_PC_ISABIT    -- EJTAG PCTrace include ISA as PC Bit0
SCAN_INSERT        -- Controls scan insertion and synthesis
```

```
SCAN_MIX_CLOCKS   -- scan chains can cross clock boundaries with
                     lock-up latches
SCAN_NUM_CHAINS   -- number of scan chains
SCAN_SCL          -- scan collar insertion on RAM interfaces
SEN_DIST          -- scan enable distribution method
SEN_BUFFERS       -- scan enable buffering
RAM_BIST_MUX      -- include test RAM mux and ports
THREAD_SCHEDULER  -- location of thread scheduler
CONTEXTS          -- Number of contexts (threads) in the processor
```

Appendix B. LX8000 Port Descriptions

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

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

1. Clocked in the JTAG_CLOCK domain.
2. Clocked in the BUSCLK domain if crossbar or LBC are asynchronous. Otherwise, clocked in the SYSCLK domain.
3. Does not require a constraint (e.g., a clock).
4. A false path (e.g. a configuration input tied to a constant).
5. Timing is specified with a symbol in techvars.scr script (e.g. RAM timing).

The table below shows the port connections for the top level module of the LX8000 processor, known as lx2. The timing information and notes have the same meaning as for the previous table.

Table 46: LX8000 Processor Port Summary

Port Name	I/O	Timing	Description
Clocking, Reset, Interrupts and Control			
SYSCLK	input	(3)	Processor clock.
ResetN	input	(4)	Warm reset (or reset "button").
CResetN	input	(4)	Cold reset (or power on).
RESET_D1_R_N	input	(4)	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_D1_BR_N	input	(4)	BUSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_PWRON_C1_N	input	(4)	Power on reset copy for JTAG.
RESET_PWRON_D1_LR_N	input	(4)	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	(4)	Interrupt requests.
EXT_HALT_P	input	50%	External stall line.
Configuration			

Port Name	I/O	Timing	Description
CFG_TLB_DISABLE	input	(4)	Disable TLB mappings even if pop_tlb.
CFG_SLEEPENABLE	input	(4)	Sleep enable configuration.
CFG_RAD_LEXOP[5:0]	input	(4)	LEXOP encoding. Must be 011111 for LX8000.
CFG_RAD_DISABLE	input	(4)	LEXOP disable configuration. Must be zero for LX8000.
CFG_SINGLEISSUE	input	(4)	Single issue mode configuration. Must be zero for LX8000.
CFG_HLENABLE	input	(4)	Strap to one to enable internal HI/LO registers.
CFG_MACENABLE	input	(4)	Strap to one to enable internal MAC (if present).
CFG_MEMSEQUENTIAL	input	(4)	Strap to one if line reads return words in sequential order, zero if interleave order.
CFG_MEMZEROFIRST	input	(4)	Strap to one if line reads return word zero first, zero if desired word first.
CFG_MEMFULLWORD	input	(4)	Strap to one if main memory must be written with 32-bit words, zero if byte and halfword writes are allowed.
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	(4)	Strapped with DMEM base address configuration value.
CFG_DWTOP[23:10]	input	(4)	Strapped with DMEM top address configuration value.
CFG_IWBASE[31:10]	input	(4)	Strapped with IMEM base address configuration value.
CFG_IWTOP[`23:10]	input	(4)	Strapped with IMEM top address configuration value.
CFG_IWROM	input	(4)	Strap to one to treat IMEM like a ROM. (Note, new applications should use IROM instead of ROM-like IMEM.)

Port Name	I/O	Timing	Description
CFG_IROFF	input	(4)	Strap to one to disable IROM.
CFG_DWDISW	input	(4)	Strap to one to disable processor DMEM writes. Must be zero for LX8000.
CFG_EJDIS	input	(4)	Must be strapped to zero.
Test and Debug			
JTAG_RESET_O	output	20%, (1)	JTAG is in TEST-LOGIC-RESET state.
JTAG_RESET	input	(4)	JTAG is in TEST-LOGIC-RESET state.
TAP_RESET_N_O	output	20%, (1)	TAP controller reset.
TAP_RESET_N	input	(4)	TAP controller reset.
JTAG_TDO_NR	output	50%, (1)	Test data out.
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	(4)	Test reset.
EJC_ECRPROBEEN_R	output	30%	One indicates EJTAG probe is active.
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
SEN	input	(4)	Scan Enable
TMODE	input	(4)	Test Mode Pins
SIN[<k>:0]	input	(4)	Scan Input. <k> can range from 7 to 0.
SOUT[<k>:0]	output	(4)	Scan Output. <k> can range from 7 to 0.

Port Name	I/O	Timing	Description
RBC_SEL[7:0]	input	(4)	RAM BIST RAM select code: 10000000 - instruction MEM 01000000 - data MEM 00100000 - dcache data store 00010000 - dcache tag store 00001000 - icache tag store, set 1 00000100 - icache inst store, set 1 00000010 - icache tag store, set 0 00000001 - icache inst store, set 0
RBC_WE[<k>:0]	input	(4)	RAM BIST write enable, where <k> is 1 for word write granularity, 7 for byte write granularity.
RBC_RE	input	(4)	RAM BIST read enable.
RBC_CS	input	(4)	RAM BIST select.
RBC_ADDR[15:0]	input	(4)	RAM BIST address.
RBC_DATAWR[63:0]	input	(4)	RAM BIST write data.
RBM_DATARD[63:0]	output	(4)	RAM BIST read data.
Data RAM DMA Access			
DMADW_RCLK	input	(3)	Data RAM DMA clock.
DMADW_DATAINDEX[17:4] (MAX)	input	(5)	Data RAM DMA address.
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 or Crossbar)			
LAddrO[31:0]	output	(2), 20%	Address.
LCmdO[8:0]	output	(2), 20%	Output command.
LDataO[63:0]	output	(2), 20%	Output data.
LDataI[63:0]	input	(2), 50%	Input data.
LIrdyO	output	(2), 20%	Initiator ready.

Port Name	I/O	Timing	Description
LIrdyI	input	(2), 30%	Other initiators ready.
LFrameO	output	(2), 20%	Transaction frame.
LFrameI	input	(2), 30%	Frame from other initiators.
LSelI	input	(2), 30%	Slave select.
LTrdyI	input	(2), 30%	Target ready.
LGTidO[15:0]	output	(2), 20%	LBC global thread ID.
LGTidI[15:0]	input	(2), 30%	LBus global thread ID.
XBRdVld	input	(2), 30%	Crossbar read data valid.
XBRdSize	input	(2), 30%	Split read data size.
SpltRdFull	output	(2), 30%	Read data queue full.
LId	output	(2), 20%	Instruction/data.
LUc	output	(2), 20%	Bus request.
LCoe[9:0]	output	(2), 20%	Command output enable.
LToe	output	(2), 20%	Transaction output enable.
LDoe[7:0]	output	(2), 20%	Data output enable.
LReq	output	(2), 50%	Bus request.
LGnt	input	(2), 30%	Bus grant.
Shared RAM Request/Grant Interface			
EXT_IWREQRAM_R	input	30%	External hardware drives to one to request access to IMEM.
IW_GNTRAM_R	output	30%	Cpu drives to one to grant external IMEM access request.
EXT_DWREQRAM_R	input	30%	External hardware drives to one to request access to DMEM.
DW_GNTRAM_R	output	30%	Cpu drives to one to grant external DMEM access request.
EXT_ICREQRAM_R	input	30%	External hardware drives to one to request access to ICACHE.
IC_GNTRAM_R	output	30%	Cpu drives to one to grant external ICACHE access request.
EXT_DCREQRAM_R	input	30%	External hardware drive to one to request access to DCACHE.
DC_GNTRAM_R	output	30%	Cpu drives to one to grant external DCACHE access request.
Coprocessor Interface			
C<z>condin	input	80%	Cop branch flag.
C<z>rd_addr[4:0]	output	50%	Cop read address.

Port Name	I/O	Timing	Description
C<z>rhold	output	45%	Cop hold condition, one stalls coprocessor.
C<z>rd_gen	output	50%	Cop general register read command.
C<z>rd_con	output	50%	Cop control register read command.
C<z>rd_data[31:0]	input	80%	Cop read data.
C<z>wr_addr[4:0]	output	20%	Cop write address.
C<z>wr_gen	output	20%	Cop general register write command.
C<z>wr_con	output	20%	Cop control write address command.
C<z>wr_data[31:0]	output	30%	Cop write data.
C<z>invld_M	output	60%	Cop invalid instruction flag, one indicates invalid instruction in M stage.
C<z>xcpn_M	output	60%	Cop exception flag, one indicates exception in M stage.
C<z>rd_cntx[2:0]	output	40%	Cop read context number.
C<z>wr_cntx[2:0]	output	30%	Cop write context number.
C3cnt_iparet	output	20%	Count instructions retired Pipe A
C3cnt_ipbret	output	20%	Count instructions retired Pipe B
C3cnt_ifetch	output	20%	Count instruction fetches
C3cnt_imiss	output	20%	Count icache misses
C3cnt_istall	output	20%	Count icache stalls
C3cnt_dmiss	output	20%	Count dcache misses
C3cnt_dstall	output	20%	Count dcache stalls
C3cnt_dload	output	20%	Count data load operations
C3cnt_dstore	output	20%	Count data store operations
Event Control and Thread Observation			
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.

Port Name	I/O	Timing	Description
Block Transfer Engine			
RXT<z>_CTL_R	input	30%	Receive TBus Control (1=control beat, 0=data beat).
RXT<z>_DATA_R[63:0]	input	30%	Receive TBus Data.
RXT<z>_RDY_R	output	30%	Asserted when new receive descriptor is available.
TXT<z>_RDY_R	input	30%	Asserted when transmit scheduler is ready to accept transmit data.
TXT<z>_INCSEQ_R	input	30%	Asserted when transmit scheduler wants BTEs to increment their sequence number.
TXT<z>_DONE_R	input	30%	Asserted when TBus will be idle in the next cycle.
TXT<z>_BUSY_R	output	30%	Asserted when the BTE is transmitting data. Deasserted 2 cycles before finishing.
TXT<z>_CTL_R	output	30%	Transmit TBus Control (1=control beat, 0=data beat).
TXT<z>_DATA_R[63:0]	output	30%	Transmit TBus Data.

Appendix C. LX8000 Pipeline Stalls

This section documents stall conditions that may arise in the LX8000.

C.1. Stall Definitions

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

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

Stall: if not otherwise qualified, means pipeline stall.

C.2. Instruction Groupings

These instruction groupings are used to describe stall conditions that are based on the type of instructions in the pipeline.

Table 47: 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-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)
M-I-UnlinkedBranch	BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ
M-I-General	All remaining M-I instructions.
MIV-CMove	MOVZ ,MOVN
EJTAG-Control	DERET, SDBBP, M16SDBBP

C.3. Non-Sequential Program Flow Issue Stall

M-I JR, JALR:
Two issue stalls after the delay slot instruction.

M-I J, JAL, 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.

C.4. Load Subword Stall

Load instructions which have Byte or Halfword operands always cause a one-cycle stall.

C.5. Store-Load Stall

A Load instruction which follows a Store instruction by one cycle causes a one-cycle stall if the Store instruction hits in the Dcache or has a Byte or Halfword operand.

C.6. StoreAny - StoreSubword Stall

If the LX8000 is configured to work with RAMs that have word write granularity, a Store instruction which has a Byte or Halfword operand, and which follows any Store instruction by one CYCLE, always causes a one-cycle stall. Alternatively, the LX8000 can be configured to work with RAMs support byte write granularity, which eliminates the stall.

C.7. Load/Store Ops Stall Matrix

The following table summarizes the stall rules related to Load and Store instructions described above. In this table, the "2nd OP" refers to an instruction which issues in the CYCLE after the "1st OP".

Table 48: Load/Store Ops Stall Matrix

	1st OP			
2nd OP	M-I, LW, LT	M-I, LB(U), LH(U)	SB, SH	SW
non load-store	-	1U	-	-
LW, LB(U), LH(U))	-	1U	1W	1U
SB, SH	-	1U	1W	1U
SW	-	1U	-	-

Notes: - means no stalls
xU indicates unconditional stall for the indicated number of cycles
xS indicates stall only if 2ndOp Source = 1stOp Load-target
xW indicates stall if data RAMs have word-write granularity

C.8. MVCz Stall

The coprocessor move instructions (M-I: LWCz, MTCz, MFCz, and MTLXC0, MFLXC0) are always followed by a single cycle issue stall.

C.9. IMMU Stall

When the program jumps, branches, or increments between the two most recently used pages, a single cycle stall is incurred.

When the program jumps, branches or increments to a third page a two-cycle stall is incurred.

C.10. IMMU Issue Stall

When an IMMU 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 IMMU stalls:

M-I branch of any kind
M-I J, JAL
EJTAG DRET

C.11. Icache 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.

C.12. Dcache 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.

When a data cache miss occurs as a result of a load byte or load halfword, the processor stalls for the duration of the cache line fill operation.

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

C.13. Pipeline Timing Diagrams for Stalls

C.13.1. Non-Sequential Program Flow Issue Stalls

M-I JR,JALR

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

M-I J, JAL, and M-I taken branches

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

M-I not-taken branches

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

C.13.2. Load Subword Stall

```
lb             I  D  S  E  M  M  W
foo2              I  D  S  E  E  M  W
foo4                 I  D  S  S  E  M  W

RHOLD                      X
```

C.13.3. Store-Load Stall

```
sw s0,4(a0)    I  D  S  E  M  W
lw s2,0(a0)       I  D  S  E  M  M  W
foo3                 I  D  S  E  E  M  W

RHOLD                        X
```

C.13.4. StoreAny - Store Subword Stall

```
sw s0,4(a0)    I  D  S  E  M  W
sb s2,0(a0)       I  D  S  E  M  M  W
foo3                 I  D  S  E  E  M  W

RHOLD                        X

sh s0,4(a0)    I  D  S  E  M  M  W
sb s2,0(a0)       I  D  S  E  E  M  M  W
foo2                 I  D  S  S  E  E  M  W

RHOLD                      X     X
```

C.13.5. MVCz Stall

```
mtc0        I  D  S  E  M  W
foo            I  D  D  S  E  M  W
foo1              I  D  S  E  M  W
```

C.13.6. LWCz Stall

```
lwc0        I  D  S  E  M  W
foo            I  D  D  S  E  M  W
foo1              I  D  S  E  M  W
```

C.13.7. Icache Miss Stall

```
foo0          I  D  S  E  M  M  M  M  M  M  W
foo2             I  D  S  E  E  E  E  E  E  M  W
foo4                I ~d  .  .  .  I  D  S  E  M  W

RHOLD                     X  X  X  X  X
```

C.13.8. Dcache Miss Stall

```
lw            I  D  S  E  M  .  .  .  .  W
foo2             I  D  S  E  M  M  M  M  M  W
foo4                I  D  S  E  E  E  E  E  M  W

RHOLD                        X  X  X  X
```