LEXRA

NetVortex Data Sheet

Lexra, Inc.

Release 1.9

April 2, 2001

NetVortex Data Sheet Revision 1.3, for RTL Release 1.9.

Table of Contents

List of Tables

List of Figures

1. NetVortex Product Overview

1.1. Introduction

This data sheet describes NetVortex, a scalable, multi-processor architecture developed specifically for use in network communications systems. NetVortex employs Lexra's LX8000 processor, and incorporates significant architectural features to support emerging network communications applications. A multi-processor IC based on NetVortex can perform IP routing and classification tasks at data rates up to OC-192 (30M packets/second).

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 13). The major subsystems are: the CPU core, Local Memory Interfaces (LMI) and LBus Controller (LBC). The technology includes an optional interface to a customer-defined Coprocessor (CI2) and optional customer extensions to the MIPS ISA (Custom Engine). The local instruction memories and data memories may include caches and fixed RAM; 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 Block Transfer Engine (BTE) attached to a dedicated RAM port, that is used in NetVortex 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. NetVortex is a scalable multi-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. NetVortex 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 Block Transfer Engine (BTE) connected to each 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, NetVortex 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.

A typical system design based on NetVortex is illustrated in Figure 2 on page 14, which shows 16 processors, that include local instruction and data storage (not shown). A high bandwidth crossbar connects the processors to shared devices such as TCAMs, SRAMs, and custom logic. A Device Management Interface allows an application-specific management processor to access the shared devices.

NetVortex provides two optional peripherals: the Test and Set Engine and the Block Transfer Controller. The Test and Set Engine attaches to the crossbar and supports up to 32 unique semaphores. These semaphores may be used to control access to resources shared among any of the processors and contexts that have access to the semaphores. The Block Transfer Controller is connected to a dedicated port of each processor's data memory and transfers packet data over dedicated busses to external ports. IC designs using NetVortex can cost-effectively support a wide spectrum of network communications systems.

NetVortex employs Lexra's LX8000 packet processor, which is an extension Lexra's LX4189 processor. The LX8000 incorporates the LX4189's 6-stage RISC pipeline. As a result NetVortex can achieve high system clock performance in a portable cell-based design. The 6-stage pipeline also decouples customer-configurable RAMs from critical paths internal to the core.

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 assembler code.

This data sheet describes the base LX4189 processor as well as LX8000 extensions to the LX4189. The remaining sections of this data sheet describe: hardware and instructions that support context switching (Section 2); the general RISC programming model (Section 3); LX8000 instruction extensions, including instructions for context switch, and bit-field processing (Section 4); processor memory interfaces (Section 5); coprocessor interfaces (Section 6); embedded debug support (Section 7); the NetVortex crossbar interconnect between processors and shared devices, and the device interface protocol. (Section 8); the optional NetVortex Test and Set Engine (Section 9); and the optional NetVortex Block Transfer Controller (Section 10)

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, configurable sizes.
    - Local data memory, configurable sizes.
    - Memory interface logic included.
    - Crossbar interface to access system devices.
    - Split read transactions over crossbar interface.
    - Optional customer-defined coprocessor.
    - Optional customer-defined instruction extensions
      support EJTAG Draft 2.0 with extensions for multi-thread debugging.

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

- **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.
- **Block Transfer Controller**
    - Performs block transfer between processors and external interfaces.
    - Supports one or two Utopia-4 receive/transmit pairs, at 415 MHz, 32 bits.
    - Fully integrated with each processor's local data RAM.
    - Can handle up to 4 simultaneously active transfers.
    - Internal busses move up to 256 bits of data per cycle.
    - Internal data bandwidth is 115 Gbits/s at 450 MHz.
    - Maintains packet ordering.
- **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.
- **Supports up to 16 LX8000 Packet Processing Engines**
    - High-speed RISC CPUs optimized for packet processing applications.
    - A 16-processor system provides a performance of 7200 MIPS at 450 MHz.
    - Can process all seven networking protocol layers.
- **Crossbar Bus Architecture**
    - High bandwidth transfer paths between processors and shared devices such as SRAMs and TCAMs.
    - 28.8 Gbits/s of read bandwidth and 28.8 Gbits/s of write bandwidth per device at 450 MHz.
    - 28.8 Gbits/s of read bandwidth and 14.4 Gbits/s of write bandwidth per processor at 450 MHz.
- **The Test & Set Engine**
    - Optional crossbar-attached device.
    - Supplies up to 32 unique semaphores.
    - Controls access to shared resources.
- **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**
  - Sizteen processors, each with 4 contexts, 16KB IMEM, 16KB DMEM.
  - Block Transfer Controller.
  - TSMC 0.15µm. results (typical process, worst case operating conditions):
    Clock: 450 MHz
    Area: 64 mm^2.
    Power: 6.8 W
- **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.
- **Ultimate Scalability**
  - A single LX8000 processor for a SOHO VPN (for example)
  - Up to 16 LX8000 processors for an OC-192 router.
- **EJTAG Debug**
  - Optional extension the EJTAG 2.0.0 specification
  - Supports multi-processor and 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.

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

The figure below shows the structure of the LX8000 processor, as used in NetVortex.

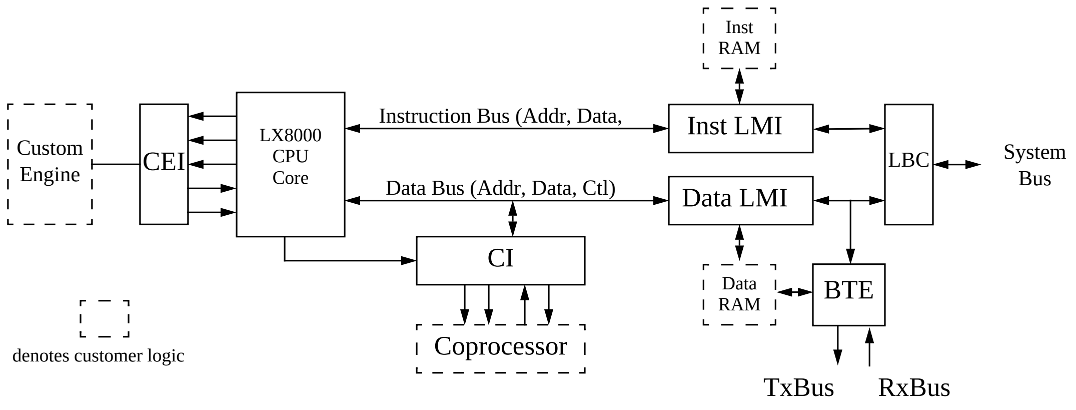

Figure 1: NetVortex LX8000 Processor Overview

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/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.

Block Transfer Engine. In a NetVortex system, the LX8000 processor includes a dedicated Block Transfer Engine (BTE) that provides efficient high-bandwidth packet transfer between the processor's local data RAM (DMEM) and the NetVortex Utopia-4 interfaces. Software initiates transfers with Write Descriptor instructions (WD) that pass a transfer descriptor to the BTE. The BTE can hold eight transfer descriptors per thread, consisting of up to four Rx descriptors and up to four Tx descriptors.

1.4. NetVortex System Overview

The NetVortex system includes up to 16 LX8000 processors, and adds packet transfer pathways, shared device interfaces and a crossbar to provide high-bandwidth, low-latency communication between the processors and shared devices.

The figure below shows the structure of the NetVortex system, which uses the LX8000 processor as a building block.

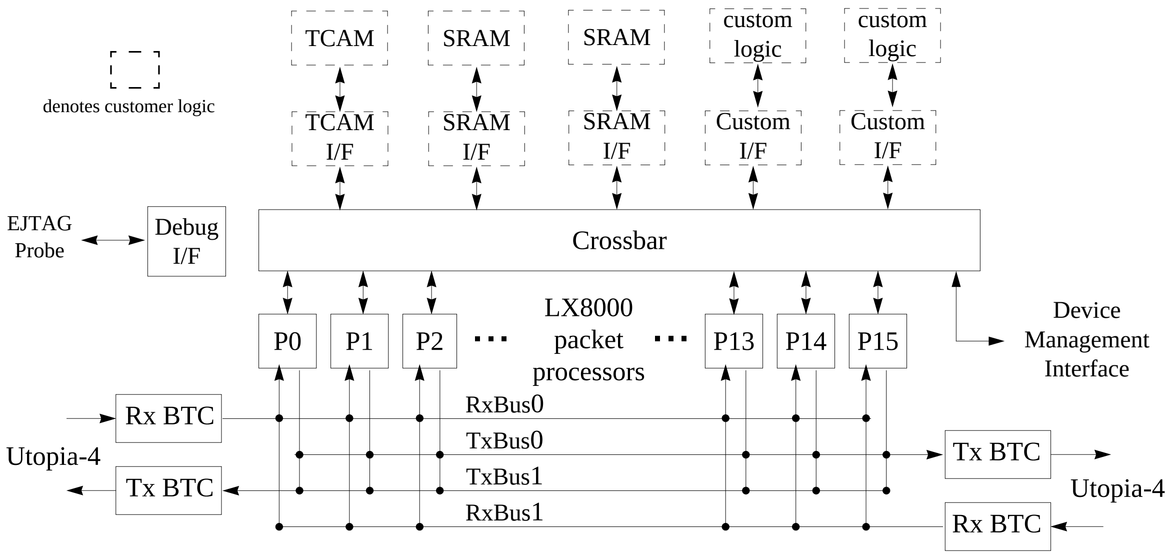

Figure 2: NetVortex System Overview

Block Transfer Controllers. The Block Transfer Controllers (BTCs) are responsible for moving data between each processor's BTE and the external Utopia-4 Receive (Rx) and Transmit (Tx) ports. The BTC's Utopia-4 interfaces operate at 415 MHz and are 32 bits wide.

Crossbar Interconnect. The crossbar operates at 450 MHz and supports simultaneous full duplex data transfer between the processors and device interfaces. The crossbar provides a sustainable data transfer bandwidth of 58 Gbits/sec per device interface.

Each processor can pass one 64-bit write or read request to the crossbar every two cycles. Device interfaces accept write or read requests every cycle, and can source 64 bits of read data every cycle. The crossbar incorporates queues that are dedicated to each device and processor to prevent head-of-line blocking. Arbitration is performed by independent per-queue arbiters. Each arbiter implements windowed round-robin selection. The LX8000 processor context switches are used in a processor to hide the latency of transactions that are performed over the crossbar.

Shared Device Interfaces. The crossbar connects the processors to shared device interfaces, that in turn connect to on-chip or off-chip resources. These interfaces are shared among the processors and provide the bandwidth and flexibility required for a wide spectrum of applications.

Device Management Interface. The crossbar's optional Device Management Interface (DMI) provides a port for accessing the crossbar devices. This port may be used by a management processor or other application specific logic.

NetVortex 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^{TM}) for popular pure-play foundries.

Section 1.5 describes the building blocks, and Section 1.6 describes the deliverable models.

1.5. 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).

- One optional Coprocessor Interface (CI) per processor.

- A Local Memory Interface (LMI) supports instruction RAM (IMEM) and data RAM (DMEM).

- A Lexra Bus Controller (LBC) to connect peripheral devices and secondary memories to the processor's own local buses.

NetVortex employs multiple LX8000 processors and additional modules to provide a complete system-on-a-chip for high performance packet processing.

- Block Transfer Controllers (BTCs) with Utopia Level 4 interfaces coordinate the transfer of packets between external data path components and internal DMEM.

- Crossbar device interfaces for attachment of application-specific TCAMs, SRAMs, and network co-processors.

- Device Management Interface (DMI) to for management processor access to shared devices.

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

1.5.1. 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.

1.5.2. 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.

1.5.3. Coprocessor Interface

Lexra supplies an optional Coprocessor Interface (CI) for applications requiring this functionality. 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.

1.5.4. Custom Engine Interface

The NetVortex 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.

1.5.5. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the interface between the LX8000 and shared devices attached to the crossbar. On the processor side, the LBC provides a command buffer of configurable depth to prevent processor stalls. On the crossbar side, the LBC provides a configurable-depth queue for read data.

1.5.6. Block Transfer Controllers

The Block Transfer Controllers (BTCs) move data between LX8000 data memory (through the BTE) and the external Receive (Rx) and Transmit (Tx) ports.

The Rx BTC and Tx BTC modules pass data and control information between the external interfaces and the internal buses. The external interfaces support Utopia Level 4 at 415 MHz with 32-bit data bus. Each of the two RxBuses and two TxBuses are 64 bits wide. They provide an aggregate internal transfer bandwidth of 115 Gbits/sec at 450 MHz.

Support for SPI Level 4 Phase 2 is planned for future releases.

1.5.7. Crossbar Device Interfaces

The crossbar connects the processors to shared device interfaces, that in turn connect to on-chip or off-chip resources such as TCAMs, SRAMs, and network-specific coprocessors. These resources are shared among the processors and provide the bandwidth and flexibility required for a wide spectrum of applications.

Store instructions, Load instructions, and Write Descriptor (WD) instructions are used to control transactions between the processors and shared devices. The interfaces support atomic read-modify-write operations, enabling devices to implement advanced functions such as statistics and metering.

1.5.8. Device Management Interface

The crossbar’s optional Device Management Interface (DMI) provides a port for accessing the crossbar devices with a management processor or other application specific logic. The DMI provides support for all crossbar operations to the devices. Through the DMI, a management processor can read or modify the shared device contents; for instance, to update routing tables or poll statistics memory.

1.5.9. Building Block Integration

The NetVortex 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 NetVortex core.

1.6. RTL Core & SmoothCore

Lexra delivers NetVortex 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 NetVortex.

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, NetVortex 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.

1.7. 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.

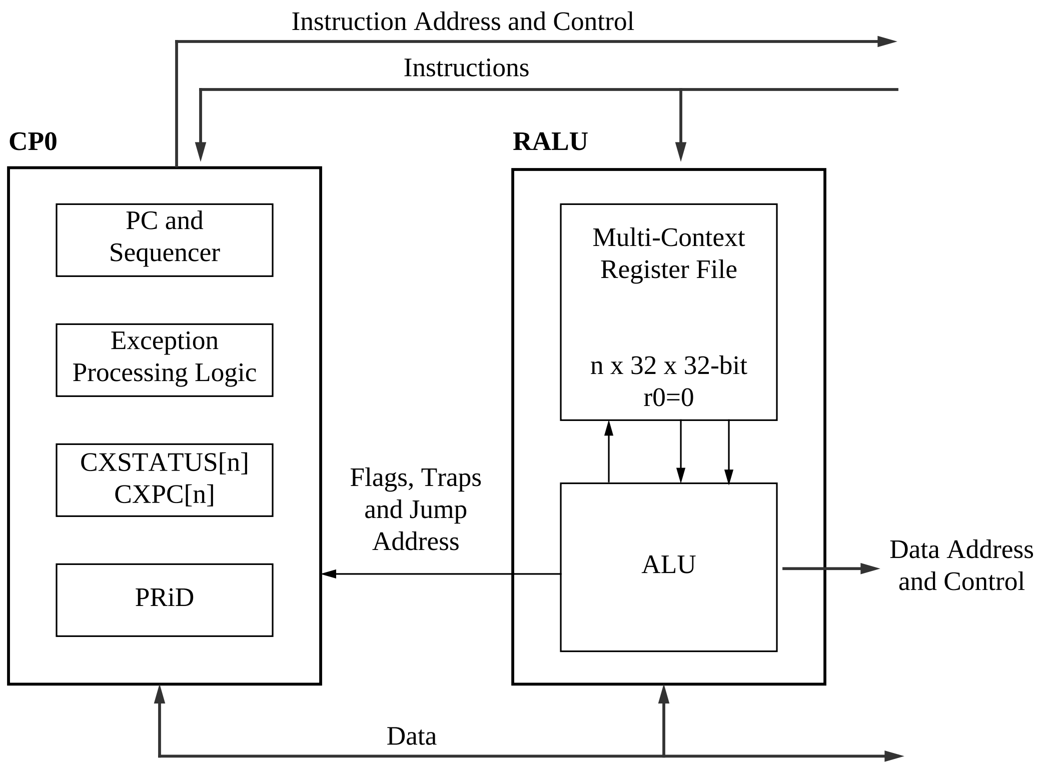
Instruction Address and Control
Instructions
CP0
RALU
PC and Sequencer
Exception Processing Logic
CXSTATUS[n] CXPC[n]
PRiD
Multi-Context Register File
n x 32 x 32-bit r0=0
Flags, Traps and Jump Address
ALU
Data Address and Control
Data

Figure 3: 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.

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 an optional application-specific Coprocessor. 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 4. The currently active context number is an implicit read-only value that is accessed with the MYCX instruction.

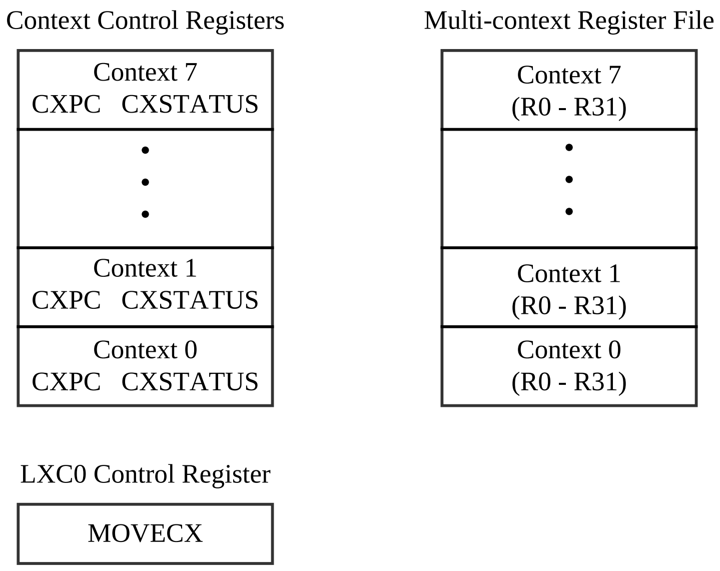

Figure 4: Context Associated Registers

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 optional NetVortex Block Transfer Controller (BTC) is an example of a system bus device that responds to the WD* family of instructions. When the BTC is present in the NetVortex system configuration, two of the eight WAIT-EVENT bits are dedicated to monitor its completion signalling.

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.

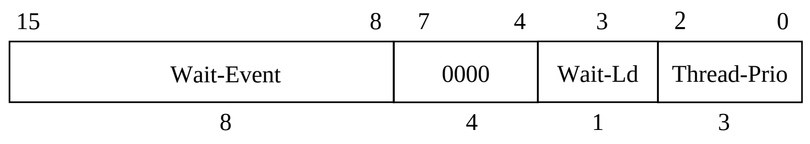

Table 3: Context Status Register Detail

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.

In NetVortex, the two-cycle bus tenure needed to issue a split read request applies only to the processor's crossbar interface. Within the crossbar and at the device interface, a split read request and 64-bit data return each require only one cycle of tenure. The return of 128-bit data requires two cycles of tenure at the crossbar interface.

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. For example, the WD.CSW instruction is used to start block transfers using the optional Block Transfer Controller supplied with NetVortex.

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 21, 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 Section 4, LX8000 Instruction Extensions.

3.1. Summary of MIPS-I Instructions

The NetVortex executes MIPS-I instructions as detailed in the tables below. To summarize, the NetVortex 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 rD, rA, rB ADDU rD, rA, rB ADDI rD, rA, immediate ADDIU 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 rD, rA, rB SUBU 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 rD, rA, rB ANDI 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 rD, rA, rB ORI 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 NetVortex 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 NetVortex Core. Each is synchronized by the Core to the NetVortex 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 NetVortex.

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 NetVortex. 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 6, 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. For cross-processor notification, the optional Test and Set Engine may be used as described in Section 9, NetVortex Test and Set Engine.

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 <u>differences</u>, these are indicated with <u>underlined text</u>.

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 5, 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.

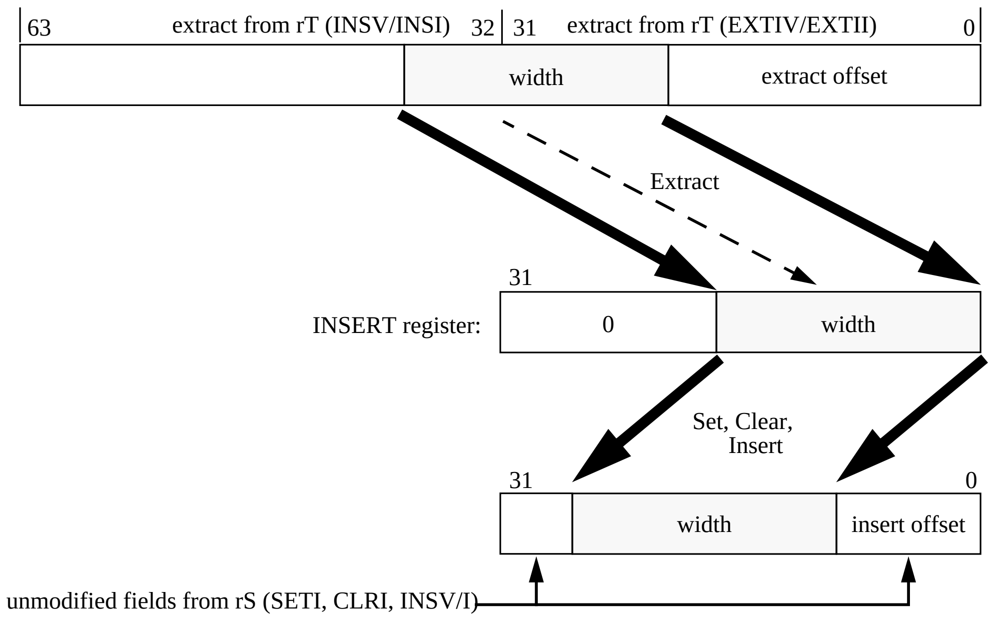
63
extract from rT (INSV/INSI)
32
31
extract from rT (EXTIV/EXTII)
0
width
extract offset
Extract
31
INSERT register:
0
width
Set, Clear, Insert
31
0
width
insert offset
unmodified fields from rS (SETI, CLRI, INSV/I)

Figure 5: 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	EXTII rD, rT, width, offset The *offset* is a value n in the range 0-31. The *width* is a value m in the range 1-32 (which is encoded in the instruction as a 5-bit value modulo 32 — that is, the value 32 is encoded as zero). These *parameter* fields are saved in the implied register INSERT. Considering rT to contain the least significant 32 bits of the extraction source, a 32-bit intermediate extraction value *temp* is generated as follows: 1) if (n+m-1) < 32, (least significant word only) the bits rT[m+n-1:n] are copied into *temp*[m-1:0] and the remaining bits of *temp* are set to zeroes. 2) if (n+m-1) > 31, (straddle two words) the bits rT[31:n] are copied into *temp*[31-n:0] and the remaining bits of *temp* are set to zeroes. The *temp* value is stored in rD and also saved in the implied register INSERT.
Insert Bits Immediate	INSI rD, rS, rT, offset This instruction must be coded as the next sequential instruction in the program sequence after an EXTII. Otherwise, its results are unpredictable. All exceptions are inhibited for the execution of this instruction. This includes hardware interrupts, debug exceptions and address exceptions. The parameter fields m, n and the intermediate extraction value *temp* are taken from the implied register INSERT, as described for EXTII. The *offset* is a value p in the range 0-31. Considering rT to contain the most significant 32 bits of the extraction source, the final extracted value *result* is generated as follows: 1) if (n+m-1) < 32, the bits *temp*[31:0] are copied into *result*[31:0]. 2) if (n+m-1) > 31, the bits *temp*[31-n:0] are copied into *result*[31-n:0]. The bits rT[n+m-33:0] are copied into *result*[m-1:32-n]. The remaining bits of *result* are set to zeroes. The bits from *result*[m-1:0] are copied into rD[m+p-1:p]. The remaining bits of rD are copied from the corresponding bits of rS. If m+p is greater than 32, the results are unpredictable.
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 6, 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.

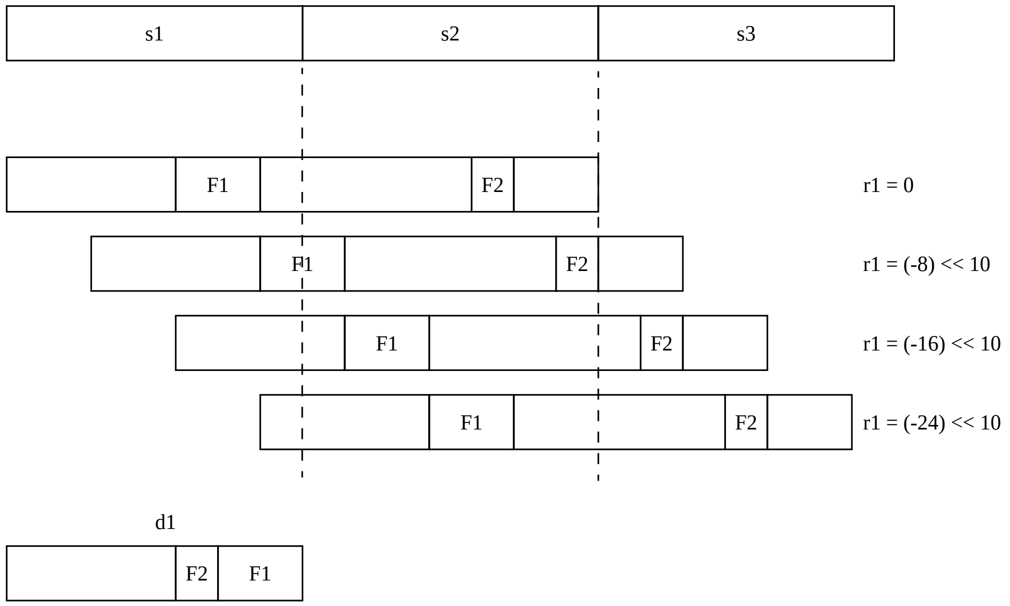

Figure 6: Packet Field Compaction with Variable Alignment

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

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

Keysize	KeyBit	Source Bits	Keysize	KeyBit	Source Bits	Keysize	KeyBit	Source Bits
15	14	29 13	**16**	15	20 4	**17**	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
	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 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
18	17	29	**19**	18	14	**20**	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
	7	24 8		8	16 0		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
21	20	3	**22**	21	26	**23**	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
	9	25 9		10	10		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[2:0]							
Inst[5:3]	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 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 NetVortex 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 LMIsconnect 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 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.

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 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. Note that the LX8000 used in NetVortex does not include an instruction cache or data cache.

Table 23: Local Memory Interface Modules

Name	Description
IMEM	Instruction RAM.
DMEM	Data RAM or ROM.

5.2. Cache Control Register: CCTL

CCTL. Coprocessor 0 General Register Address = 20

31-6	5	4	3-0
Reserved	IMEMOff	IMEMFill	Reserved

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 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.

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 5.3. 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. To use the IMEM again, an application must re-initialize the IMEM contents through the IMEMFill bit of the CCTL register.

5.3. 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 5.2, Cache Control Register: CCTL. The IMEM module services instruction fetches that falls within its configured range. The IMEM is a convenient, low-cost alternative to a cache that makes instruction memory available to the core for high-speed access.

The configurations supported by IMEM, and the synchronous RAMs required for each, are summarized in Table 24.

Table 24: 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 25 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 24.

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

Table 25: IMEM RAM Interfaces

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

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*. 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.

5.4. 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.

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.

The configurations supported by the DMEM, and the synchronous RAMs required for each, are summarized in the Table 26.

For NetVortex, 128-bit wide dual-port RAMs are used, and the second port is connected to the processor's Block Transfer Engine (BTE).

Table 26: 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	128 x 64 bits	64 x 128 bits
2K bytes	256 x 64 bits	128 x 128 bits

Configuration	DMEM_DATA RAM (64-bit)	DMEM_DATA RAM (128-bit)
4K bytes	512 x 64 bits	256 x 128 bits
8K bytes	1,024 x 64 bits	512 x 128 bits
16K bytes	2,048 x 64 bits	1,024 x 128 bits
32K bytes	4,096 x 64 bits	2,048 x 128 bits
64K bytes	8,192 x 64 bits	4,096 x 128 bits
128K bytes	16,384 x 64 bits	8,192 x 128 bits
256K bytes	32,768 x 64 bits	16,384 x 128 bits

Table 27 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 26.

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

Table 27: DMEM RAM Interfaces

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

6. 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.

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

A coprocessor may contain up to 32 general registers and up to 32 control registers. Each of these registers is up to 32 bits wide. Typically, programs use the general registers for loading and storing data on which the coprocessor operates. Data is moved to the coprocessor's general registers from the 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).

6.2. 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 28: 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 NetVortex system clock.

6.3. 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.

6.4. 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
```

6.5. 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
```

6.5.1. 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.

6.5.2. 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.

7. LX8000 EJTAG

7.1. 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[1] 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.

7.2. 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 NetVortex 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 NetVortex. 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.

1. With the internal PC trace buffers available in NetVortex, the full EJTAG feature-set can be accessed through just 4 JTAG pins - TCK, TDI, TDO and TMS.

- 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.

7.2.1. 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. In NetVortex with internal PC trace buffers it is not used.

Table 29: 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 30: 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 31: EJTAG Synthesis Constraints[2]

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

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

7.3. Single Processor PC Trace

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

7.3.1. 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.

7.3.2. 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.

7.3.3. 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).

7.3.4. Single-Processor PC Trace Pinout

Table 32: 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 33: 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

7.3.5. 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 NetVortex from all other exceptions, a 4-bit code is used instead.

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 NetVortex can be configured via the EJTAG_XV_BITS lconfig option to use only the 3-bit standard codes. In that case, if a vectored interrupt is taken, the 3-bit code for RESET will be presented.

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

7.3.6. 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.

7.4. Multiprocessor EJTAG

In the multiprocessor case Lexra's EJTAG solution enables independent, simultaneous control of each processor with the same pinout as the single-processor case. Each processor has its own EJTAG block and TAP controller. The TAP controllers are daisy-chained together such that the TDO from the first processor is connected to the TDI of processor 2, and the TDO of processor 2 is connected to the TDI of processor 3. The TDO of the last processor on the chain is connected to the EJTAG probe.

7.4.1. Connectivity Requirements

In order to allow seamless operation with external probes, Lexra's EJTAG TAP controller chain must have no register stages between it and the external probe. Any multiplexing of the external pins must be set such that there is a direct connection to and from the Netvortex TAP controller chain while using EJTAG.

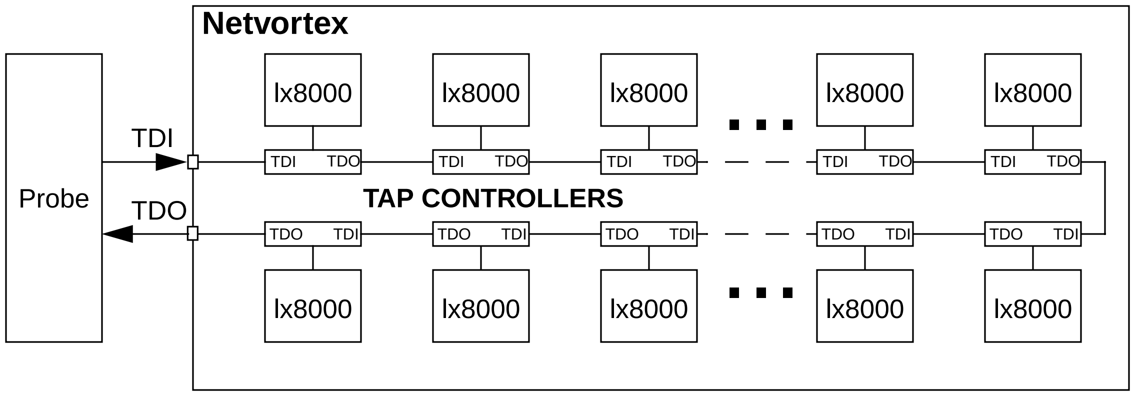

Figure 7: Construction of Chained TAP controllers for Multiprocessor EJTAG

7.4.2. Multiprocessor PC Trace Using Internal Trace Buffers

In multiprocessor systems, traditional PC trace solutions become impractical due to the number of pins needed for each processor and the mismatch between processor and probe speeds. By using internal PC trace buffers no external PC trace pins are required, providing full multiprocessor EJTAG and PC trace capability using just 4 pins. The buffers allow real-time compressed PC trace information for each processor to be buffered on-chip and scanned out serially using existing JTAG pins at probe clock speeds. The program flow can then be reconstructed externally to give an accurate history of the execution of programs running on each processor.

The PC Trace buffer keeps a history of all non-sequential changes to the PC such as branches, jumps and exceptions. On each non-sequential PC change a buffer entry is written which contains the current PC, the current context, whether or not a trigger point occurred since the previous frame and optionally two characteristic counters - the number of sequential instructions and the number of stall cycles since the previous buffer entry. The size of the sequential instruction count field is configurable from 2 to 8 bits. The stall cycle count field is also configurable and is split into 2 parts - mantissa and exponent. Both mantissa and exponent can have widths of 0 to 4 bits. If the mantissa width is set to zero the stall cycle count field is removed from the buffer.

The buffer width ranges from 36 to 58 bits depending on the amount of bits allocated to the characteristic counters and the number of contexts. The depth of the buffer is a trade-off between the amount of history required and the amount of die are available. It is configurable in powers of two from 16 frames upwards with a typical value of 64 frames.

8. NetVortex Crossbar Interconnect

The crossbar interconnect prevents processor-device communication from becoming a system bottleneck. The crossbar network supports a subset of the full LBus transaction types:

- Write (processor to device).
- Split read request (processor to device).
- Write with split read request (processor to device)
- Split read data (device to processor).

The crossbar does not support line read transactions. When used with the crossbar, packet processors may not include instruction cache or data cache. The processor uses 64-bit split reads to fill the instruction RAM (IMEM). The processor also converts all non-split load instructions to split loads. No context switch is performed for converted transactions, and the processor is stalled until the load completes. This transparent mode of operation is intended to be used for system initialization and other non-critical tasks.

Split read transactions that are generated by converting non-split transactions are referred to as f*oreground* transactions. Split read transactions that are generated through the normal mechanisms are sometimes referred to as *background* transactions.

The crossbar does not support EJTAG DMA operations. However, software running on the probe may still access devices by jamming the appropriate instructions into a packet processor.

The organization of the crossbar network is centered around a 16 processor, 6 device configuration. A special interface is also included to allow an additional (possibly non-NetVortex) processor to access devices that are attached to the crossbar.

General characteristics of the 16 processor, 6 device system are:

- Processor and device interfaces support full duplex operation. That is, transactions can pass simultaneously in both directions.
- A processor may pass one write, split-read or write-split-read request to the crossbar every two cycles. These are converted to single cycle transactions within the crossbar network.
- Processors service split data responses at the maximum rate of one every two cycles.
- A device may respond to more than one device ID. For all other purposes, a multi-ID device appears to the crossbar as a single device.
- Devices operate as Write Descriptor targets, or memory storage devices. The two modes of operation may not be mixed within a single device.
- Devices may generate and accept transactions at the rate of one per cycle.
- There is no processor-to-processor or device-to-device traffic in the system.
- Transactions involving a given processor and given device are delivered in order. There are no other ordering guarantees.

8.1. Processor-to-Device Paths

Figure 8 below illustrates the processor-to-device crossbar network. The numbers just above each queue indicate which device(s) may be targeted by transactions held in the queue.

- The first column of queues receive write, split-read and write-split-read requests from each processor, and prevents threads from blocking while a request waits for a path through the crossbar interconnect. These queues are implemented in the processor's LBC.
- The connection to the crossbar uses a modified LBC interface that allows the processor to simultaneously source a transaction and receive split data resulting from an earlier request.
- The 4x6 crossbars sort the requests from the per-processor queues into per-device queues.
- A sub-unit of four processors, one 4x6 crossbar and associated queues makes a convenient tile used for the design of a 16 processor system.
- If each processor attached to the 4x6 crossbar issues a request every two cycles and requests are evenly distributed among 6 devices, there is an 88% probability that a request will be accepted by the crossbar. A request every four cycles will result in a 94% probability of acceptance. (See below for the source of these figures).
- From the crossbar output queues, 5:1 muxes transfer requests to devices. With adequate queue depth at the crossbar outputs, the devices can be fully utilized.
- Uniform behavior is provided throughout the crossbar network. That is, if processors are performing similar work, they will experience similar delays through the crossbar.

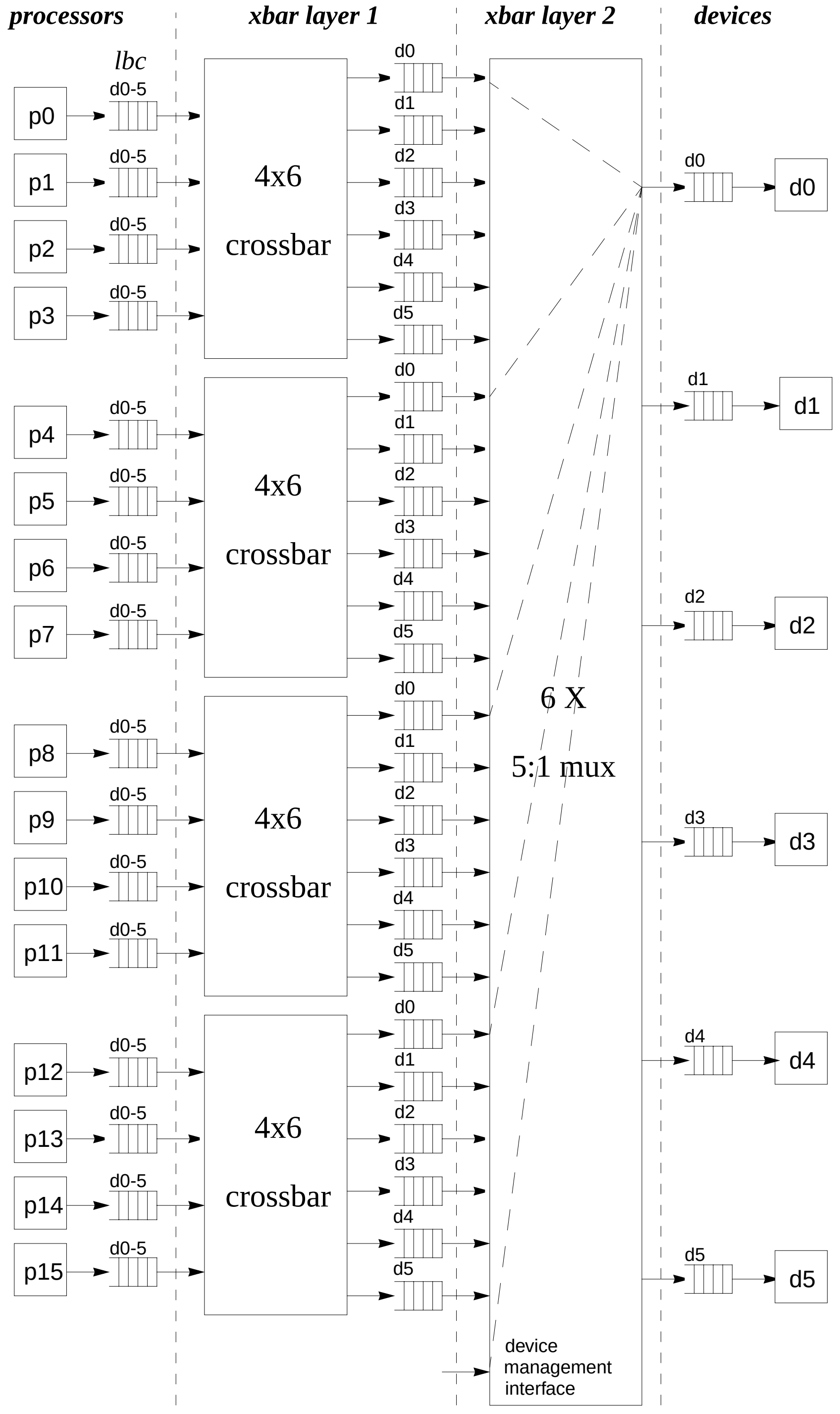

Figure 8: Crossbar for 16 Processors and DMI to 6 Devices

The probability of request acceptance quoted above is obtained from the formula[1]

$$BW = M \times \left(1 - \left(1 - \frac{pReq}{M}\right)^{N}\right)$$

where

BW	aggregate bandwidth of crossbar, in transfers per cycle
M	number of devices
pReq	probability that a processor makes a request
N	number of processors

The equation for BW is accurate to within 8% for $M/N > 0.75$. Given the bandwidth, the probability that a processor's request is accepted by the crossbar is

$$pAcc = \frac{BW}{N \times pReq}$$

8.2. Device-to-Processor Paths

Figure 9 below illustrates the device to processor crossbar network. The numbers just above each queue indicate which processors(s) may be targeted by transactions held in the queue.

- The topology is compatible with the organization of the processor-to-device network.
- The right-most column of queues are associated with the devices, and are fairly shallow.
- A simple fanout interconnect layer transfers data from the device queue to the crossbar input queue dedicated to that device. With adequate queueing, the device interfaces can be fully utilized.
- The blocking effects of the device to processor crossbar are negligible because the utilization is typically low, and the ratio of processors to devices is typically 3 to 1. Note that the simple formula given in the previous section cannot be used to estimate performance, because the M/N criteria is not met.

1. "Performance of Multiprocessor Interconnection Networks", L Bhuyan et al., *IEEE Computer*, Feb. 1989

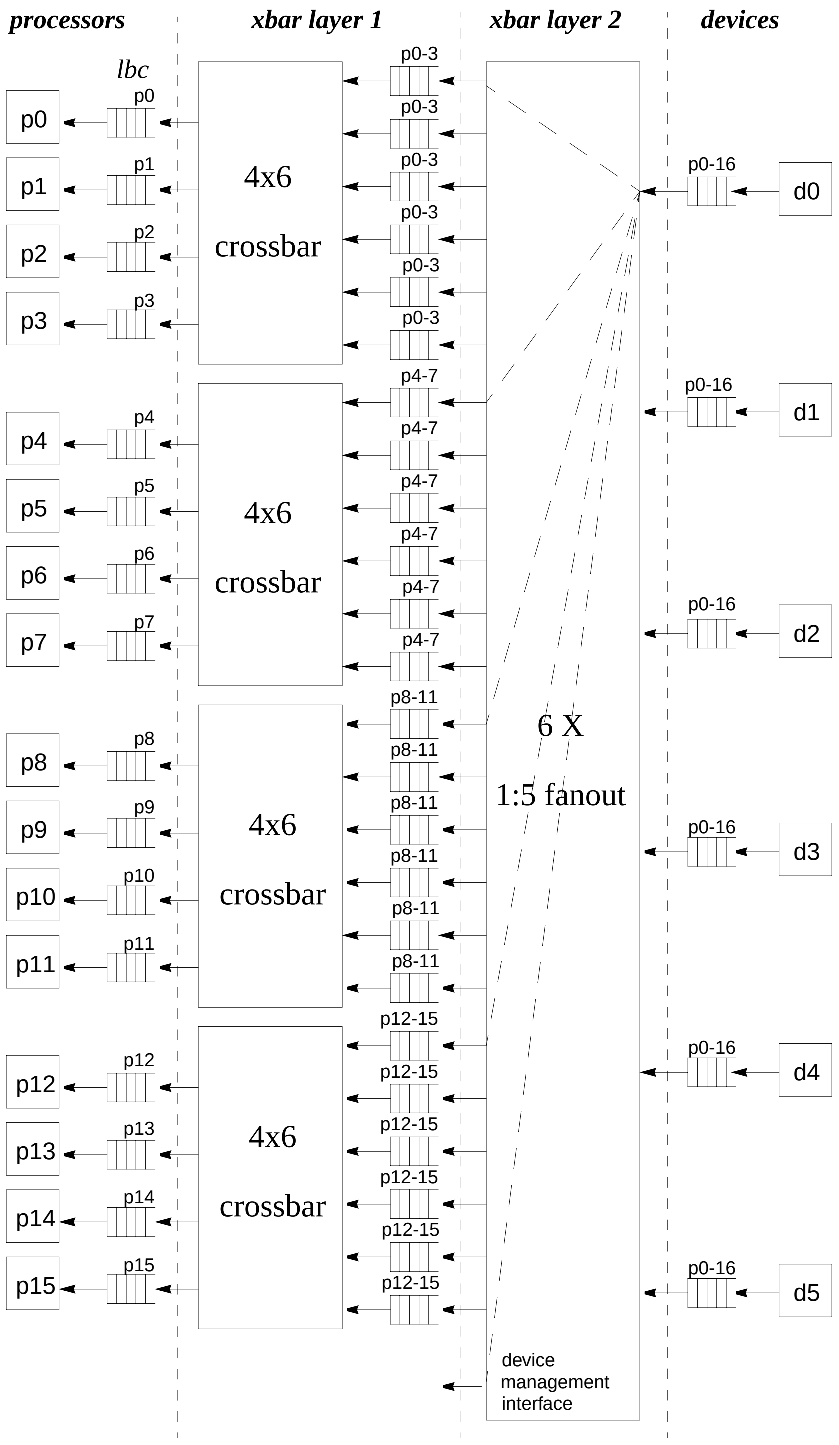

Figure 9: Crossbar for 6 Devices to 16 Processors and DMI

8.3. Bandwidth and Latency

The crossbar network can service *N***freq* requests per second directed from the processors to the devices, where *N* is the number of devices, and *freq* is the operating frequency of the crossbar. Split data return transactions are "free" with the crossbar, because they use full duplex return paths. The crossbar operates at

250 MHz in 0.15μm technology. Assuming 1 in 5 transactions is a split read request, the crossbar delivers about 10 times total processor-device bandwidth of a single shared bus, with much lower latencies.

The timing diagram below illustrates the flow of a single transaction from a processor to a device, assuming an idle system and synchronous operation.

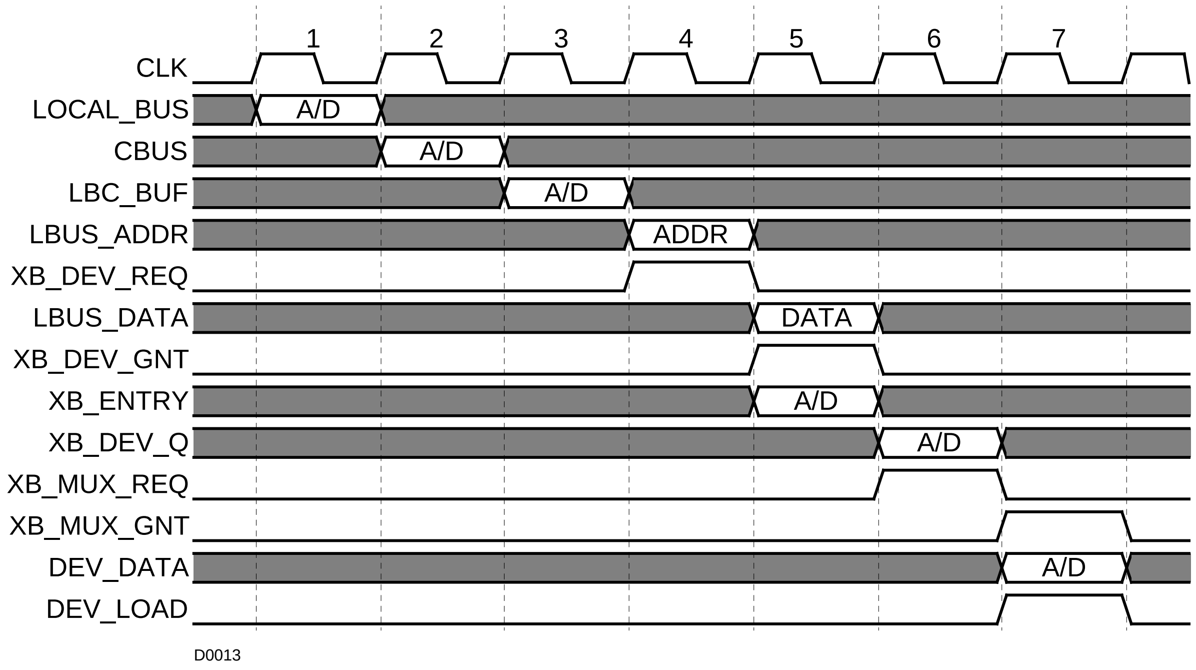

Figure 10: Processor to Device Transaction Flow

Cycle Description:

1. Local Bus - request and data are sent from the processor to the cache controller via the Local Bus.

2. CBus - request and data are sent from the caches to the LBC via the CBus.

3. LBC Buf - one cycle to go through the transaction queue in the LBC. This queue is the LBC write buffer. All transactions to the crossbar go through the LBC write buffer.

4. LBC Addr/Crossbar Device Request - first cycle on the LBus, address is sent to crossbar interface. Address is decoded and request is sent to crossbar arbiters (one per device). Requests are registered in arbiters.

5. LBC Data/Crossbar Device Grant - second cycle on the LBus, data is sent to the crossbar interface. Device Grant is given and merged Address/Data is sent across the crossbar to a crossbar device queue.

6. Device Queue/Crossbar Mux Request - Crossbar entry is seen on queue output and request to crossbar mux arbiter is sent. Request is registered in the arbiter.

7. Mux Grant - Crossbar Mux arbiter gives grant and sends load signal to device. New transaction is loaded into device.

The timing diagram below illustrates the flow of a single transaction from a device to a processor, assuming an idle system synchronous operation.

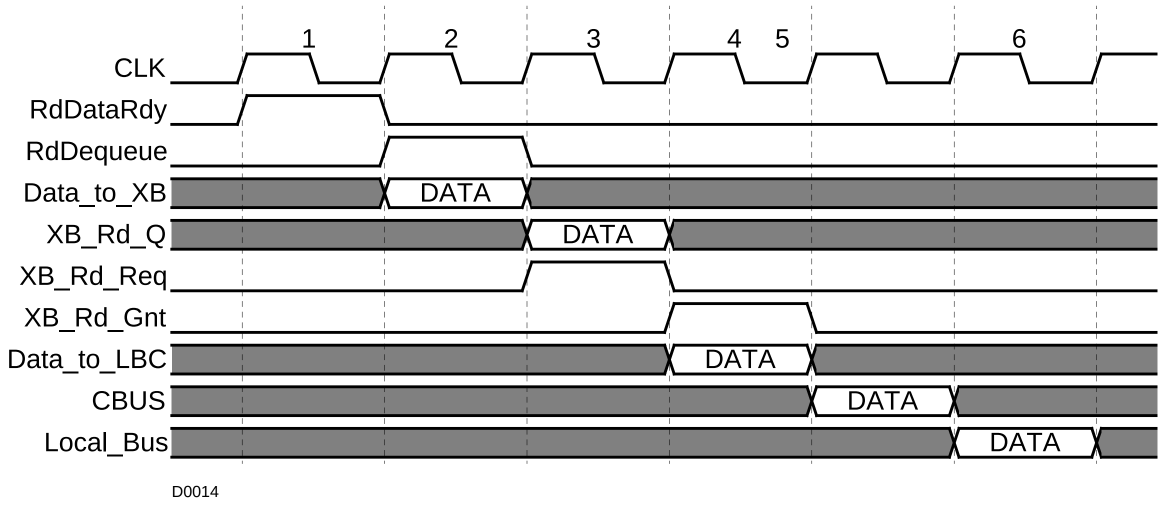

Figure 11: Device to Processor Transaction Flow

Cycle Description:

1. Device asserts RdDataRdy to send split data to processor.
2. Crossbar interface asserts RdDequeue to accept the split data. Split Data transferred to Read Queue in crossbar.
3. Crossbar Read Queue asserts request to transfer read data across crossbar to processor interface.
4. Read Grant is given and Data is dequeued from crossbar read queue and sent to LBC.
5. Data sent from LBC to cache interface via CBus.
6. Data returned to processor via Local Bus. NOTE: The local bus is 32 bits wide. Two cycles are needed to return the data for a twinword read.

8.4. Crossbar Port Configuration

The interconnect supports configurations of 1, 2, 3, 4, 5 or 6 devices, and 4, 8, 12 or 16 processors, plus a port that provides host access to the devices. The RTL is optimally reduced to support the configured number of processors and devices.

8.5. Address Decoding

The crossbar interface accepts all requests from a processor and decodes the address to determine which device the request is targeting. The decode is based on the *lconfig* options chosen for each device.

There are two types of devices supported on the crossbar - Memory mapped and write descriptor.

8.5.1. Memory Mapped Devices

When a device interface is configured as a memory mapped device, an address window is specified using *lconfig*. The window is described with a base address and mask. The upper 12 bits of the address are configurable, allowing for a minimum window size of 1 Mb. The address mask is also a 12-bit value and allows the window size to range between 1 MBytes and 2 GBytes. The crossbar uses this base address and mask to compare with the upper 12 bits of the request address to determine if it is within the address window for that device. There is only one address window for each memory mapped device interface.

8.5.2. Write Descriptor Devices

NetVortex supports up to 32 write descriptor devices. Each device port configured to be used as a write descriptor interface has an associated 32-bit device ID mask to indicate which device ID(s) are accepted by the device attached to that port. More than one bit in the ID mask may be set, allowing a single device to respond to multiple device IDs. For write descriptor requests, the crossbar uses ADDR[7:3] of the request and the device masks to determine the target crossbar device port.

All write descriptor devices are located in a single 1 MByte window. This window is selectable with *lconfig*. It cannot also be used by memory mapped devices. This results in a hole in the address space, because descriptor devices only use a small portion of this window. All memory mapped device windows and the 1 MByte write descriptor memory window must be mutually exclusive.

8.5.3. Address Error Handling

The crossbar detects two error conditions for processor initiated requests. If a request does not address a configured device, the crossbar will discard the request and assert the per-processor error flag. If a processor issues a split read request to a write-only device, the crossbar discards the request and asserts the per-processor error flag. The thread ID associated with the request is also captured. These types of errors typically occur during software debug.

The crossbar module hierarchy gathers the error signals to provide the outputs listed below. Each processor has it's own set of outputs. In the signal name, <n> represents the processor number, from 0 through 15. Customers may use these output signals as appropriate for their system level error detection and recovery strategy. For example, the error signals may be captured by a custom coprocessor that interrupts the affected processor, or the signals may be passed to centralized error reporting and handling hardware.

Table 34: Crossbar Error Reporting Signals.

Direction	Signal	Description
output	XB_ErrBadAddr<n>	Pulsed high for one cycle when the processor issues a request with an address that does not decode to a valid device.
output	XB_ErrInvldRd<n>	One bit per processor, pulsed high for one cycle when the processor issues a read request to a device that does not have a read return path.
output	XB_ErrThread<n>[3:0]	Identifies the local thread ID that caused an error signalled through XB_ErrBadAddr or XB_ErrInvldRd for a given processor.

8.6. Arbitration

Arbitration in the 4x6 crossbar unit is performed with independent per-queue arbiters. Each arbiter implements windowed, round-robin selection among the processors that compete for a given output queue.

At the next level in the interconnect, each device has the same type of independent dedicated arbiter to select one of four requests in the crossbar's per-device queues.

The same approach is used for the split data return pathways through the crossbars back to the processors.

8.7. Asynchronous Interface

An optional asynchronous boundary is implemented at the 4x6 crossbar's queues (within the crossbar layer 1 column shown in Figure 8 and Figure 9), allowing the processor and 4x6 crossbar speed to be decoupled from chip level interconnect. Implementing the async interface at this point preserves as much bandwidth as possible from the processor and through the 4x6 crossbar, compared to using the less efficient async interface in the LBC. However, this does require most of the 4x6 crossbar’s logic and local interconnect to run at full processor speed.

8.8. Queue Depths

The queues in the LBC may be sized appropriately to prevent local blocking between the processor and crossbar. The LBC’s output queues should not be used as extra overflow queuing for the crossbar when any particular device queue is full. The depth of the processor’s outgoing command queue is RTL-configurable in the range of 2 to 16 command entries. The depth of the processor’s incoming split data queue is RTL-configurable in the range of 2 to 16 twinword entries.

The LBC will stall the processor if the processor executes a write, split-read or write-split-read while the LBC output queue is full.

Because the behavior of and demand for devices may differ, each device-dedicated queue in the crossbar layer 1 (see Figure 8 and Figure 9) may be sized specifically for its device. Generally, the depth of these queues should be determined by device access patterns and crossbar performance. The depth should not be related to performance characteristics of the actual device. The FIFO depths in the first level crossbar modules are RTL-configurable for 4, 8 or 16 entries.

The devices include dedicated queues with depths that are related to overall access rates and device latency.

8.9. Instruction RAM Fill

Instruction RAMs of the packet processors are filled under the control of a hardware state machine in the processor, using split reads over the crossbar network. The state machine sequence is initiated by software with a write to the COP0 CCTL register. It is assumed that IRAM fill performance is not critical.

8.10. Device Management Interface

The crossbar’s optional Device Management Interface (DMI) provides a port for accessing the crossbar devices with a management processor or other hardware. The DMI provides support for all crossbar operations to the devices. For memory devices, this includes writes and split reads. For write descriptor devices, this includes writes and write split reads.

8.10.1. DMI Read and Write Request Interface

The table below lists the DMI signals that are used by a management processor (or similar subsystem) to transfer a write or read request to the DMI.

Table 35: DMI Request Signals

Direction	Signal	Description
mp->xb	DMI_DevReqs[5:0]	Request to individual devices
xb->mp	DMI_Gnt	Grant from crossbar, request has been taken
mp->xb	DMI_ProcNum[7:0]	Processor Number
mp->xb	DMI_Cmd[3:0]	Request Command
mp->xb	DMI_Size[2:0]	Request Size
mp->xb	DMI_ThreadID[3:0]	Request Thread ID
mp->xb	DMI_Addr[31:0]	Request Address (for memory mapped devices)
mp->xb	DMI_DevID[4:0]	Request Device ID (for WD devices)
mp->xb	DMI_Data[63:0]	Request Write Data

To send a new request to a device, the request line (DMI_DevReqs) associated with that device is asserted. Only one request line may be asserted at any time. DMI_Cmd, DMI_Size, DMI_Data, and DMI_GTid must supply valid request information. For memory mapped devices, DMI_Addr must have a valid address for that device. For Write Descriptor devices, DMI_DevID must have a valid Device ID for that device.

The crossbar asserts DMI_Gnt when it accepts the request, which will be no earlier than one cycle after the request was initiated. The crossbar uses the request information that is sampled in the cycle prior to asserting DMI_Gnt. The crossbar ignores new requests until the it has granted the current request. The interface is pipelined, allowing the crossbar to capture information for the next request, if any, while it grants the current request.

DMI_ProcNum - Processor Number

This 8-bit value is used when split read data is being returned from the devices to identify that the data is for the DMI. It must not match any of the GTID[11:4] values that are assigned to packet processors attached to the crossbar.

DMI_Cmd - Request Command

0000	<reserved>
0001	write
0010	background split read request
0011	write split read
001x	<reserved>
0110	foreground split read request
0111	<reserved>
1xxx	<reserved>

DMI_Size - Request Size

000	1 byte
001	2 bytes
010	1 word
011	2 words
100	4 word

101 - reserved
110 - reserved
111- reserved

Note: for write split read commands, the size is for the read request (either 1 or 2 words) since the write must be 2 words.

DMI_ThreadID - Request Thread ID
This value must be used to differentiate read requests when the hardware attached to the DMI can have multiple outstanding read requests. For split read requests, the value will be returned with the split read data.

DMI_Addr - Request Address
The Address is needed for all memory mapped devices. The full 32-bit physical address is sent to the device.

DMI_DevID - Request Device ID
The Device ID is used for write descriptor devices. It should be valid for any write descriptor operations.

ReqData - Write Data
Size is either 32 bits for memory mapped devices or 64 bits for write descriptor devices.

8.10.2. DMI Request Waveforms

In these waveforms, Device 0 (a value of 1 in the DMI_DevReqs) is used, but the logic is the same for any device request.

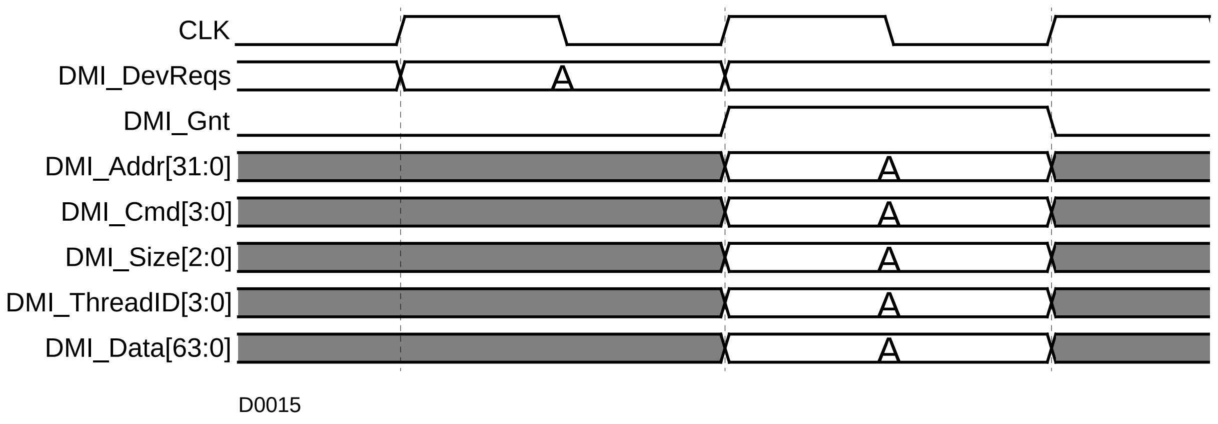

Figure 12: Single DMI Request Without Grant Delay

For all transactions, the timings for DMI_Addr, DMI_Cmd, DMI_Size, DMI_ThreadID and DMI_Data are all the same. For the waveforms below, only DMI_Addr is shown.

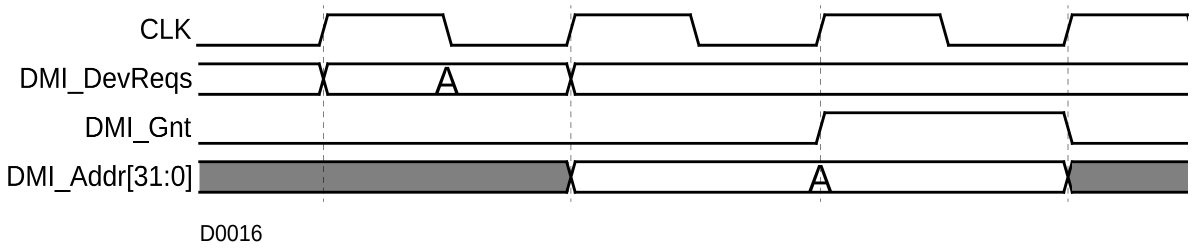

Figure 13: Single DMI Request With Grant Delay

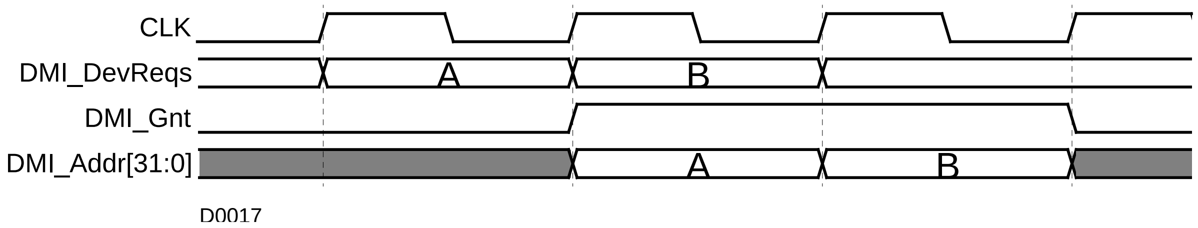

Figure 14: Back to Back DMI Requests Without Delay

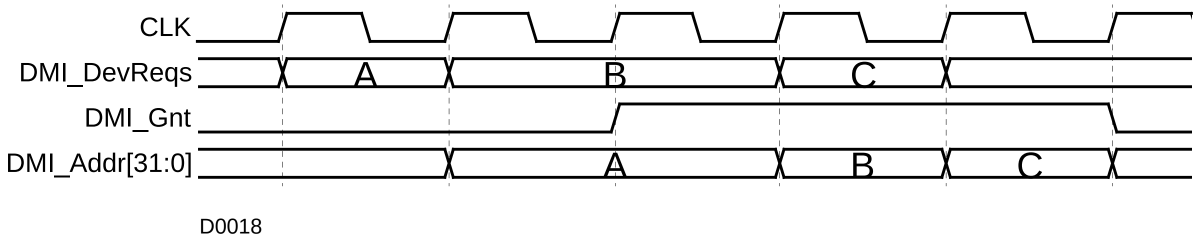

Figure 15: Multiple Back to Back DMI Requests With Grant Delay

8.10.3. DMI Split Read Data Interface

The table below lists the signals that are used by a management processor (or similar subsystem) to receive split read data from the DMI.

Table 36: DMI Split Read Data Signals

Direction	Signal	Description
xb->mp	DMI_RdValid	Split Read Data Valid
xb->mp	DMI_RdCmd[1:0]	Split Read Data Command
xb->mp	DMI_RdData[63:0]	Split Read Data
xb->mp	DMI_RdThreadID[3:0]	Split Read Thread ID
xb->mp	DMI_RdSize[1:0]	Split Read Size
mp->xb	DMI_RdBusy	Busy signal from management processor

When a split read response from a device has a Processor Number that matches DMI_ProcNum, the crossbar forwards split read the DMI interface. DMI_RdValid indicates that split read data is being sent. DMI_RdData, DMI_RdThreadID and DMI_RdSize will all have the valid split read information.

If the hardware connected to the DMI cannot accept split read data, it should assert DMI_RdBusy.

DMI_RdCmd - Split Read Data Command

00	background split read data
01	foreground split read data
1x	<reserved>

DMI_RdSize - Split Read Data Size

00	1 byte
01	2 bytes
10	1 word
11	2 words

8.10.4. DMI Read Data Waveforms

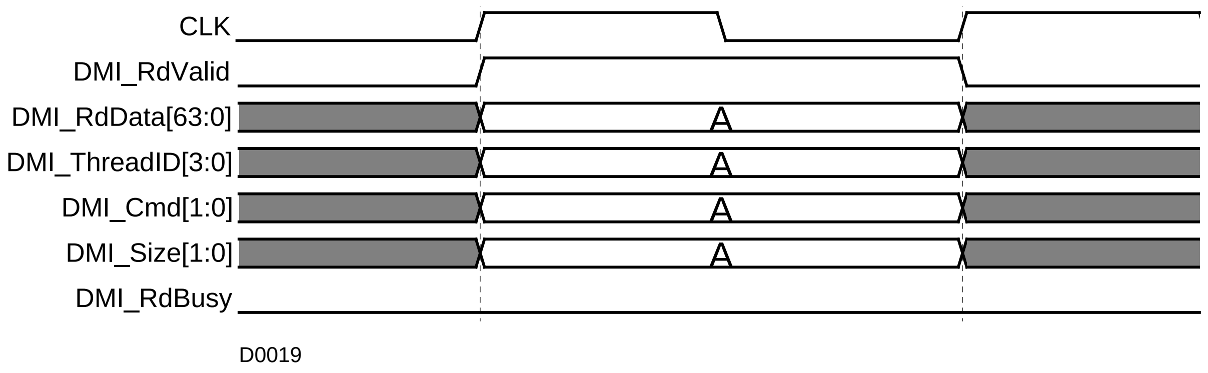

Figure 16: Single DMI Read Data Response Without Delay

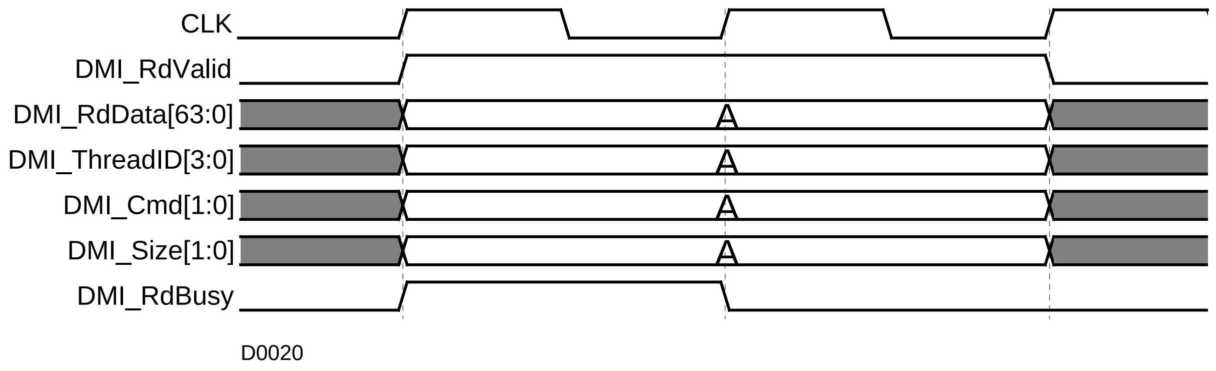

Figure 17: Single DMI Read Data Response With Busy Delay

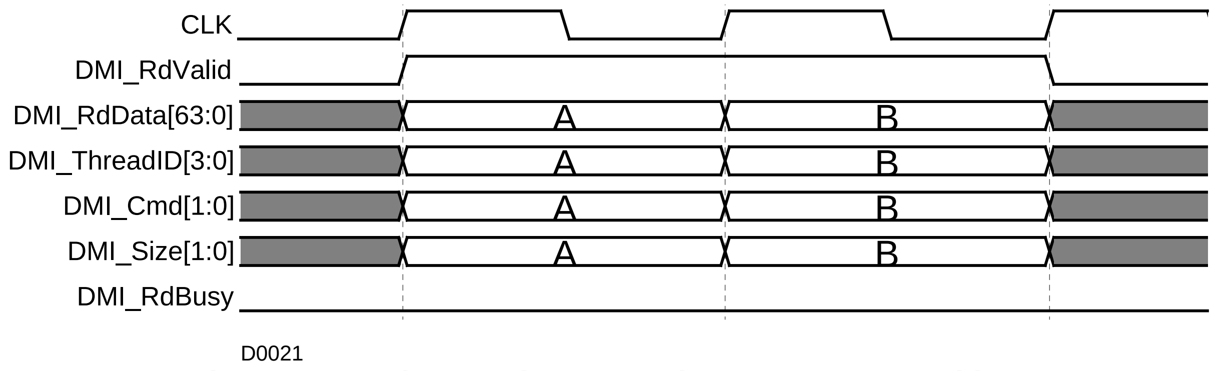

Figure 18: Back to Back DMI Read Data Response Without Delay

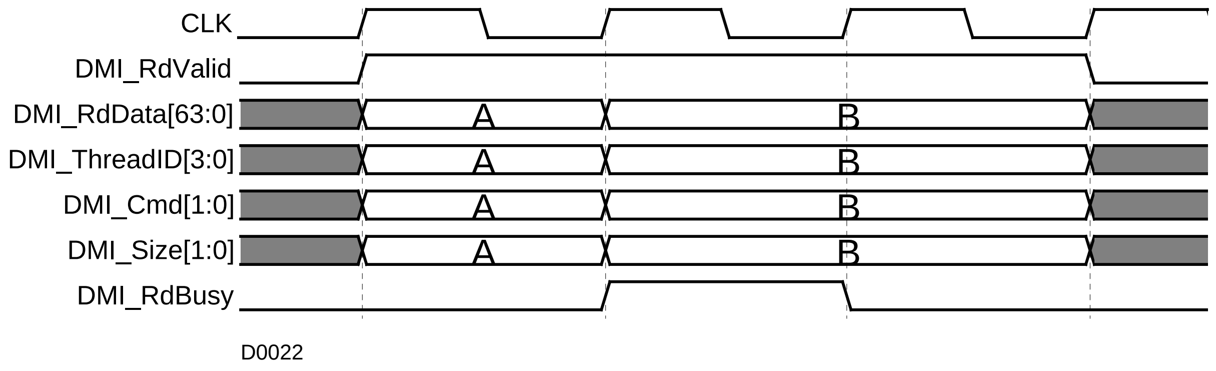

Figure 19: Back to Back DMI Read Data Response With Busy Delay

8.11.Direct FIFO Interface for Devices

The devices enqueue and dequeue information directly to and from the crossbar interconnect's queues. The devices do not implement any aspects of the LBus protocol.

8.11.1. Device Request Interface

The crossbar interface for sending requests for each device is optimized via RTL configuration. A device can either be a memory mapped device, which requires an address but supports only 32-bit writes, or a write descriptor device, which does not need an address but supports 64-bit writes for write descriptors. Each interface is configured for either memory mapped or write descriptor using *lconfig*.

Table 37: Device Request Signals

Direction	Signal	Description
xb->dev	DevReqRdy	Load new request
xb->dev	ReqCmd[3:0]	Request Command
xb->dev	ReqSize[2:0]	Request Size
xb->dev	ReqGTid[15:0]	Request Global Thread ID
dev->xb	DevBusy	Device cannot accept any more trans-actions
xb->dev	ReqAddr[31:0]	Request Address (memory mapped devices)
xb->dev	ReqData[31:0]	Write Data (memory mapped devices)
xb->dev	ReqDevID[4:0]	Write Descriptor Device ID (WD devices)
xb->dev	ReqData[63:0]	Write Descriptor Data (WD devices)

When the crossbar is ready to enqueue a new request, it asserts DevReqRdy. ReqCmd, ReqSize ReGTid, ReqAddr and ReqData are valid when DevReqRdy is asserted. If DevBusy is asserted, the crossbar cannot send a new request.

ReqCmd - Request Command

0000 - <reserved>
0001 - write
0010 - background split read request
0011 - write split read
010x - <reserved>
0110 - foreground split read
0111 - <reserved>
1xxx - <reserved>

ReqSize - Request Size

000 - 1 byte
001 - 2 bytes
010 - 1 word
011 - 2 words
100 - 4 word
101 - reserved
110 - reserved
111- reserved

Note: for write split read commands, the size is for the read request (either 1,2, or 4 words) since the write must be 2 words.

ReqGTid - Request Global Thread ID
Same as LBus Global Thread ID. For split read requests, the device should return this value with the split read data.

ReqAddress - Request Address
The Address is needed for all memory mapped devices. The full 32 bit physical address is sent.

ReqDevID - Request Write Descriptor Device ID
The Device ID is provided if more than one write descriptor device is connected to a device interface. Only available for write descriptor interfaces.

ReqData - Write Data
Size is either 32 bits for memory mapped devices or 64 bits for write descriptor devices.

8.11.2. Device Request Waveforms

(NOTE: waveforms are for memory mapped devices. Write Descriptor devices are identical except ReqAddr is replaced with ReqDevID)

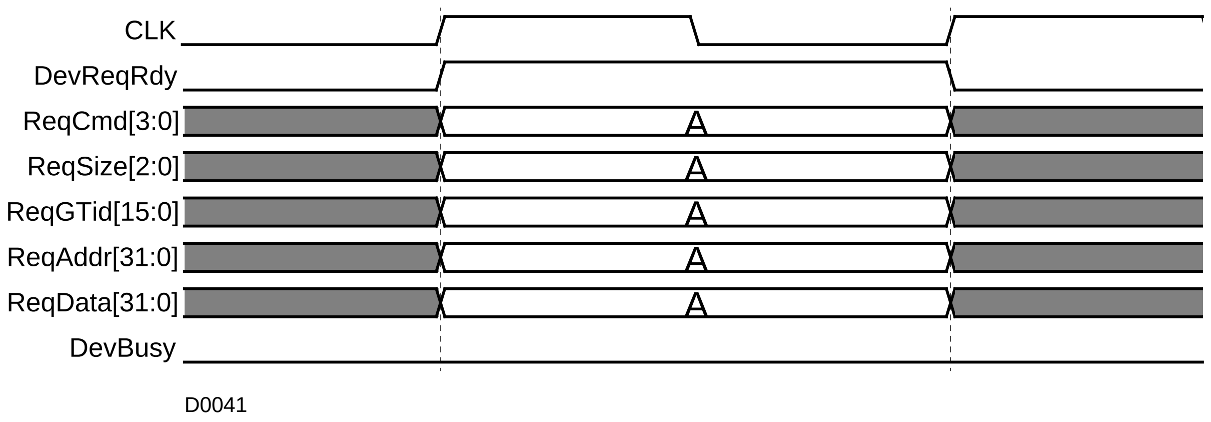

Figure 20: Single Request Enqueue

The crossbar can also issue back-to-back requests with no delay if DevBusy is not asserted.

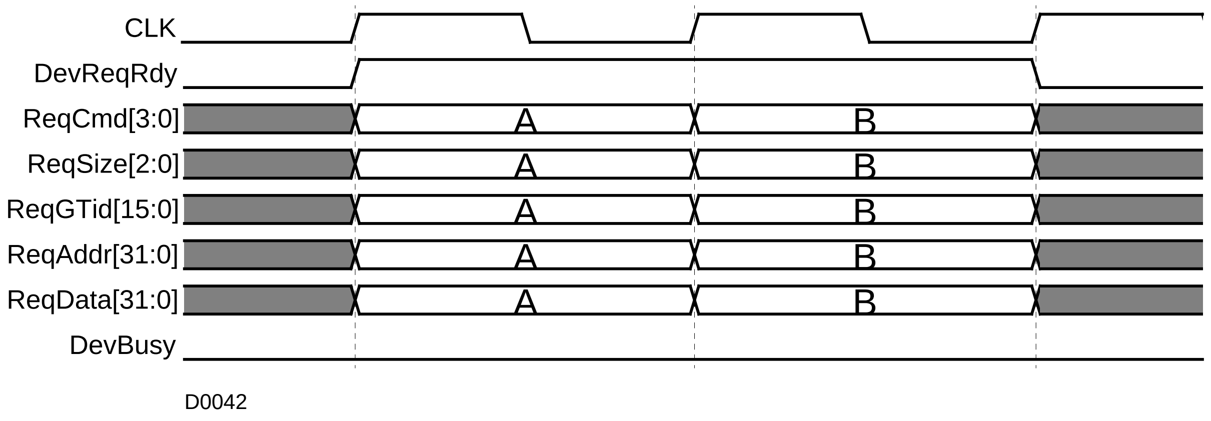

Figure 21: Back to Back Request Enqueue

DevBusy is used by the device to indicate it is not ready to receive a new request. If DevReqRdy is asserted concurrent with the assertion DevBusy, the request is ignored and must be regenerated in the next cycle.

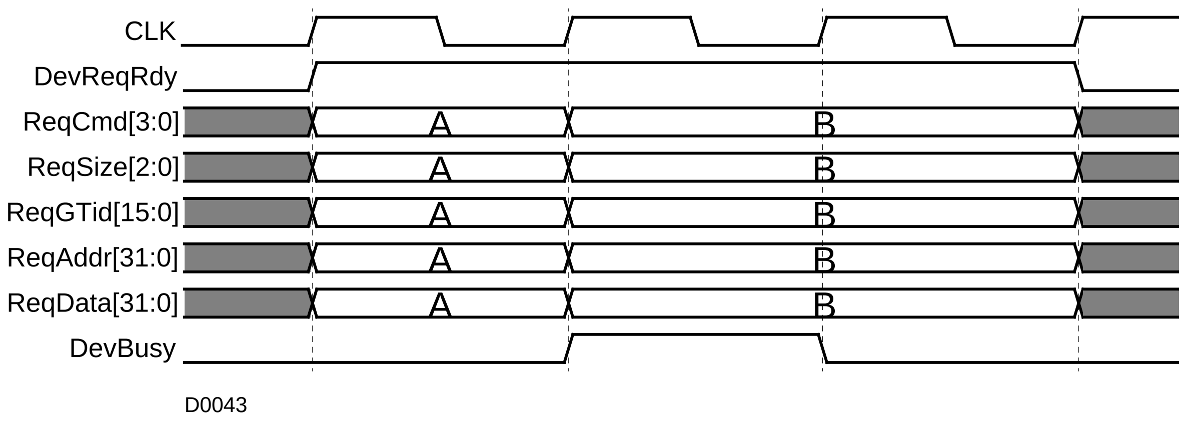

Figure 22: DevBusy Asserted Between Two Requests

8.11.3. Device Read Data Interface

The crossbar also accepts split read data to be returned to the processor. Not all devices need this interface, and each crossbar device interface can be configured using *lconfig*.

Table 38: Device Read Data Signals

Direction	Signal	Description
dev->xb	RdDataRdy	Read Data Ready
dev->xb	RdCmd[1:0]	Read Data Command
dev->xb	SplitRdData[63:0]	Read Data
dev->xb	RdGTid[15:0]	Read Global Thread ID
dev->xb	RdSize[1:0]	Read Data Size
xb->dev	RdDequeue	Crossbar accepts read data

When a device is ready to return split read data, it asserts RdDataRdy for one cycle and drives RdGTid with the GTid associated with the return data. The crossbar captures and retains RdGTid until it has dequeued the data. SplitRdData, RdCmd and RdSize are valid the cycle after RdDataRdy is asserted until the crossbar accepts the data. RdGTid and RdSize should have the same value as the split request sent on ReqGTid and ReqSize. The crossbar asserts RdDequeue when it accepts the data, and the device dequeues the entry.

RdCmd - Split Read Command
00 - background split read data
01 - foreground split read data
1x - <reserved>

DMI_RdSize - Split Read Data Size
00 - 1 byte
01 - 2 bytes
10 - 1 word

11 - 2 words

A crossbar device may need to return 4 words of data for a single request (Quadword split read). The crossbar accepts a maximum of 2 words of data per transaction, so two separate twinword operations should be used. For memory devices, the data in Addr[2] = 0 should be returned first. The GTid for both halves of the quadword data should use the same value as the quadword split read request.

8.11.4. Device Read Data Waveforms

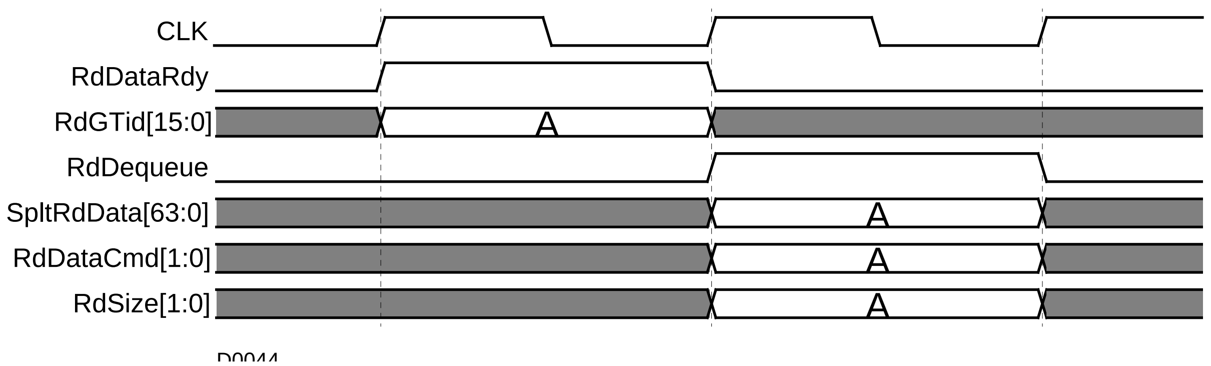

Figure 23: Single Split Read Return

Note: SpltRdData, RdDataCmd and RdSize can be driven with RdDataRdy and RdGTid in the first cycle, but the crossbar will not use them.

If the crossbar interface is not ready to accept the split read data, it will not assert RdDequeue. To avoid timing problems, the device must not use RdDequeue to gate SpltRdData, RdDataCmd and RdSize at the device outputs. Instead, these outputs are controlled by registered (next cycle) versions of RdDataRdy and RdDequeue.

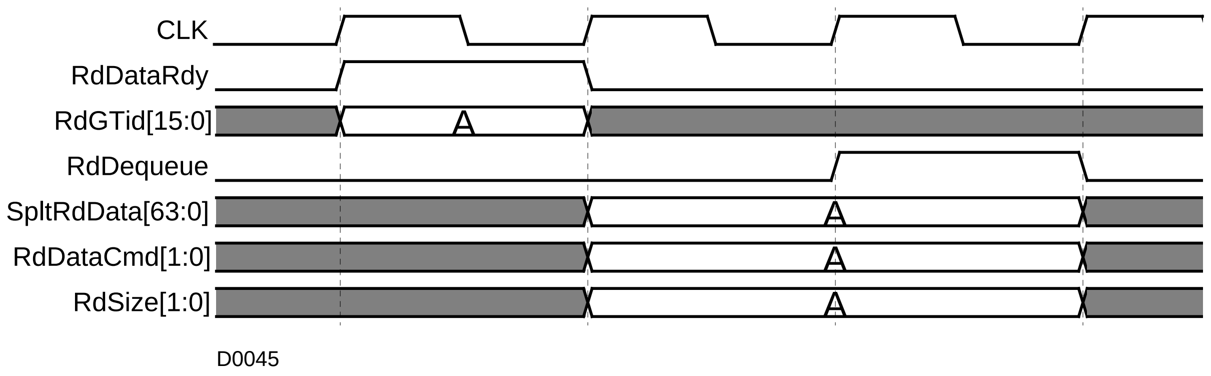

Figure 24: RdDequeue Delay

A device can issue back to back split read responses. Each RdDataRdy assertion by a device indicates a new request. If the device issues a new request while a prior request is pending (i.e., issued but not accepted by the crossbar), the device must re-issue the new request. To do so, the device leaves RdDataRdy asserted for the second request until the cycle after RdDequeue is asserted by the crossbar (i.e. the crossbar accepts the first request).

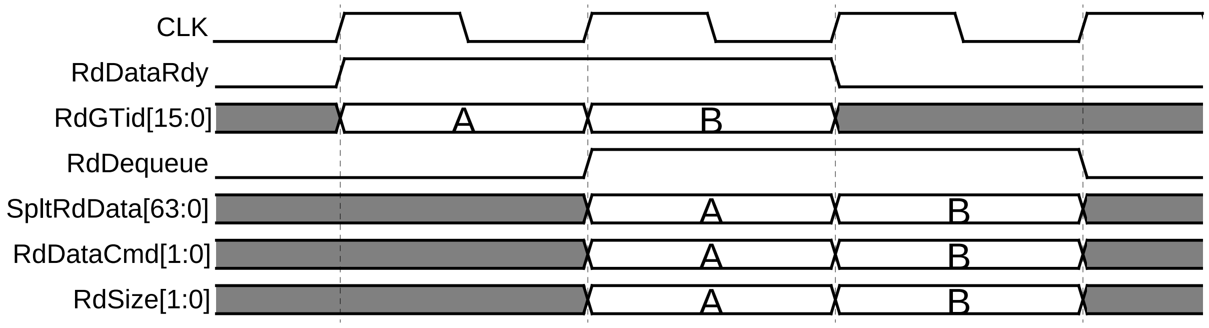

Figure 25: Back to Back Split Read Responses without RdDequeue Delay

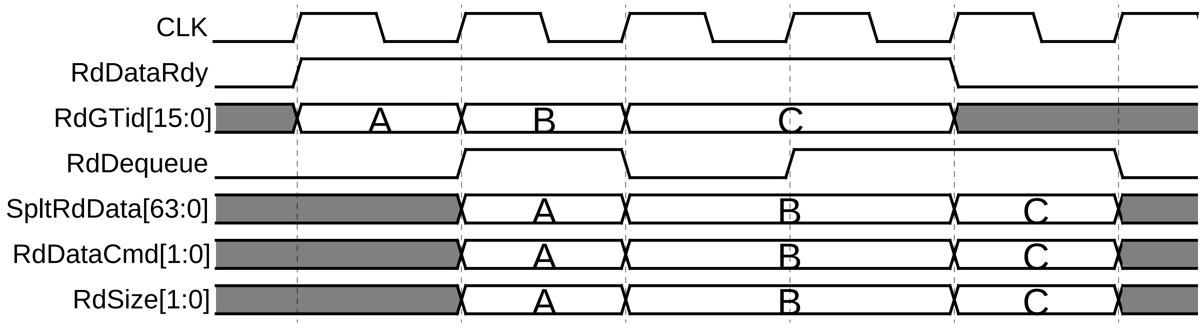

Figure 26: Back to Back Split Read Responses with RdDequeue Delay

9. NetVortex Test and Set Engine

NetVortex includes an optional Test and Set Engine. This module implements an LBus Target that supplies up to 32 unique semaphores. These semaphores can be used to control access to resources that are shared among any of the processors and contexts that have access to the LBus.

The remainder of this chapter describes the LBus commands that are supported by the Test and Set Engine as well as the resources required to implement various numbers of contexts, processors, and semaphores.

The Test and Set Engine responds to the following LBus commands:

- Single Word Read (LW instruction) — basic test and set.
- Single Word Split-Read (LW.CSW) — enqueue and wait for semaphore free.
- Single Word Write (SW instruction) — dequeue wait or clear semaphore

9.1. Semaphore Addressing

The Test and Set Engine responds to a range of addresses on the LBus that is programmable using the Lexra *lconfig* utility. Each semaphore occupies the even word of a doubleword in the address range. Up to 32 semaphores are implemented. The base address of the engine must be aligned on a 1MByte boundary, since the Test and Set Engine's configurable decoder only compares bits 31:20 of the LBus address. Address bits 7:3 identify the semaphore to be accessed and address bits 2:0 must be zero for double word addressing. Address bits 19:8 are ignored but should be zero for compatibility with future expansion.

9.2. Single Word Read — Basic Test and Set

An LBus single word read request to a semaphore address within the Test and Set Engine provides the basic functionality. If the semaphore is free, it is atomically marked as held and a value of zero is returned for the word read. If the semaphore is already held when the request arrives, it remains held and a value of one (in the least significant bit of the word) is returned for the word read.

9.3. Single Word Split-Read — (Enqueue and) Wait for Semaphore Free

An LBus single word split read request to a semaphore address within the Test and Set Engine enqueues the Processor and Context of the requestor to wait for the semaphore to be free. If the semaphore is already free, it is marked held and the return response with a value of zero is made as soon as the Engine can gain access to the LBus. If the semaphore is not free when the request arrives, the Processor and Context of the requestor is enqueued behind all others waiting for this particular semaphore (if any). Eventually, when this requestor is the oldest one waiting and the semaphore is dequeued, the return response with a value of zero is made as soon as the Engine can gain access to the LBus.

Note that the return response for a split read always has a value of zero and always leaves the semaphore in the held state.

9.4. Single Word Write — (Dequeue Wait or) Clear Semaphore

An LBus single word write request to a semaphore address within the Test and Set Engine is used to dequeue or clear the semaphore. The data associated with the write is ignored. The requestor is also ignored. (See the examples of semaphore usage below.)

If the queue for that semaphore is empty (no waiting requestors) the semaphore is marked free regardless of its previous state (free or held). If the queue of waiting requestors is not empty (the semaphore is necessarily held) the oldest requestor in the queue is removed from the queue, that requestor's return response is made as described above (in Section 7.3), and the semaphore remains in the held state.

9.5. RAM Requirements for Semaphore Queues

In order to provide a robust implementation that can withstand the worst case scenario of waiting requestors without resorting to retry mechanisms, the queues of requestors that are waiting for any of the semaphores are implemented in RAM. The size of this RAM depends on these three variables:

- Number of semaphores supported
- Number of processors in the system
- Number of contexts per processor

An expression for how the number of words in the RAM is calculated:

S = # of semaphores
P = # of processors in the system
C = # of contexts per processor

$$\text{\# 16-bit words} = 2^{(\lceil \log_2(S \cdot P \cdot C) \rceil)}$$

Table 39: Semaphore Engine RAM Requirements

number of semaphores	number of processors	contexts per processor	RAM required
4	4	4	64 x 16 bits
4	16	4	256 x 16 bits
8	4	4	128 x 16 bits
8	16	8	1K x 16 bits
16	4	4	256 x 16 bits
16	16	8	2K x 16 bits
24	4	4	512 x 16 bits
24	16	8	4K x 16 bits
32	4	4	512 x 16 bits
32	8	4	1K x 16 bits
32	16	4	2K x 16 bits
32	16	8	4K x 16 bits

9.6. Semaphore Usage for Critical Code Section

The following programming example shows typical usage of a semaphore to protect a critical section of code that should only be executed while the semaphore is held by the executing thread:

```
        la          r1,SEMAPHORE            # get semaphore address
```

```
                lw          r2,0(r1)                    # atomic test and set
                beqz        r2,CRITICAL_SECTION         # if got it, go do it
                nop                                     #   delay slot
                lw.csw      r2,0(r1)                    # else wait to get it
                nop                                     #   delay slot
        CRITICAL_SECTION:
                ...                                     # do critical section
                sw          r0,0(r1)                    # release semaphore
        END_CRIT_SECTION:
```

This example assumes that the semaphore will usually be free when it is needed. Therefore, the first check of the semaphore uses an ordinary read (lw instruction). If the semaphore is free, the critical section is entered immediately (branch around the lw.csw). If the semaphore is not free, rather than using a spin loop, which would consume both processor cycles and LBus cycles, a split read is used (lw.csw instruction) which enqueues the context in the semaphore's wait queue and allows another thread to execute. When the semaphore is later acquired, this thread becomes ready. When this thread resumes execution it immediately enters the critical section since it is guaranteed to hold the semaphore at that point.

At the conclusion of the critical section, during which the resources protected by the semaphore may be accessed, the semaphore is released with a simple write (sw instruction).

It was noted above that a write operation to a semaphore address always dequeues or clears the semaphore, regardless of the requestor or the write data. In the above coding example only the holder of the semaphore would actually clear the semaphore. Maintaining that requirement is actually no different than any other operation performed within the critical section.

9.7. Semaphore Usage for Cross Processor Wait and Post

Within one NetVortex processor a pair of threads can use the CSW and POSTCX instructions to implement Wait and Post semantics for communication between a producer thread and a consumer thread. Since the POSTCX instruction only updates the CXSTATUS register of another context within the same processor, these instructions cannot be used for communication across processors. The semaphore facility in the Test and Set Engine can be used to implement this functionality, as follows:

Let GTID1 be the Global Thread ID of the consumer thread that is ready to wait for the producer thread which has a Global Thread ID of GTID2. The following code sequences allow for cross processor Wait and Post semantics, assuming that both SEMAPHORE1 and SEMAPHORE2 have both been placed in the held state by initialization code using simple LW instructions.

In GTID, the consumer executes:

```
                ...                             # ready to wait
                la          r1,SEMAPHORE1       # get semaphore address
                la          r3,SEMAPHORE2       # clear SEMAPHORE2 to
                sw          r0,0(r3)            # indicate WAIT to GTID2
                lw.csw      r0,0(r1)            # wait for SEMAPHORE1 to be cleared
                nop                             # delay slot
                consume     ...                 # after context switch
```

In GTID2, the producer executes:

```
                produce     ...                 # ready to post
                la          r3,SEMAPHORE2
                lw          r2,0(r3)            # ensure GTID1 waiting
```

```
bnez    r2,OTHERWORK       # do something else if not
la      r1,SEMAPHORE1      # clear SEMAPHORE1 to
sw      r0,0(r1)           #   indicate POST to GTID1
```

In this example, the consumer waits for SEMAPHORE1 before consuming. The producer clears SEMAPHORE1 to post that it has produced. Note that the producer's clearing operation can take place any time after the consumer is ready to wait for the semaphore. Usually this will be after the wait begins. In the rare event that the clear occurs before the LW.CSW, the operation will still be correct even though the consumer is never enqueued for the semaphore. Also note that the LW.CSW always leaves SEMAPHORE1 in the held state for the next wait.

In case the producer is ready before the consumer, SEMAPHORE1 must not be cleared. The purpose of SEMAPHORE2 is to prevent this from happening. The consumer clears it to indicate to the producer that the consumer is about to begin waiting. If the producer finds it held, it can do some other work (or switch context — not shown). Whether the test of SEMAPHORE2 succeeds or fails, the LW instruction leaves it in the held state for the next attempted post.

9.8. Initialization

At reset time, all of the semaphores are marked free, and all of the queues are marked as empty. This does not require any RAM access, since the free-bit and queue-pointers are maintained in registers in the Test and Set Engine, rather than in the RAM itself.

10. NetVortex Block Transfer Subsystem

This chapter describes the optional NetVortex block transfer subsystem, which consists of the Block Transfer Controllers (BTC) and the Bock Transfer Engines (BTE). Section 10.1, Overview, briefly introduces the NetVortex block transfer capabilities and structure. Section 10.2, Block Transfer Buffers and Transfer Descriptors, defines the data structures that are used in conjunction with block transfers. Section 10.3, Example Transaction Flow, provides a brief description of how block transfer transactions flow through the NetVortex system. Section 10.4, Detailed Description of Block Transfer Modules, describes the elements of the block transfer subsystem.

Summary of block transfer subsystem:

- Operates at core processor speed.
- Peak transfer bandwidth of 256 bits per cycle.
- Transfer Engines dedicated to each processor.
- One or two receive/transmit port pairs for external connection.
- RAM interfaces can sustain simultaneous input/output transfers.
- Per-thread transfer descriptor queues maintained by each Transfer Engine.
- Centralized scheduler maintains optimal packet flow and packet order.

10.1. Overview

The block transfer subsystem moves blocks of data between external receive and transmit ports and the local data RAMs that are attached to one or more NetVortex processors. Threads set up transfer operations using the Write Descriptor (WD) instruction that indicates the transfer details.

The block transfer subsystem is composed of the modules listed below. Refer to Figure 27, Organization of the Block Transfer Controller.

- Block Transfer Engines (BTEs) are attached to each processor’s dual-port data RAMs.
- Per-thread descriptor queues within each BTE provide storage for receive and transmit requests that are made by processor.
- Receive DMA (RxDMA) and Transmit DMA (TxDMA) controllers within each BTE support concurrent receive and transmit transfers for each processor.
- Block Transfer Controllers (BTCs) coordinate receive and transmit traffic for a receive/transmit port pair.
- Utopia-4 receive and transmit ports are connected to the Rx BTC and Tx BTC modules. Support for SPI Level 4 Phase 2 is planned for future releases.
- Internal busses (RxBus and TxBus) provide pathways between the BTEs and the BTCs.

The Block Transfer Engine (BTE) is controlled with the Write Descriptor instructions (WD and WD.CSW) that pass a 64-bit descriptor from the CPU to the BTE. The BTE saves the descriptors in queues that may hold multiple entries per thread. There is one Rx queue and one Tx queue per thread. Writing a descriptor to the BTE does not in itself cause the transfer to start. Rather, each BTE uses information in the descriptor, in conjunction with information provided by the Block Transfer Controllers (BTCs), to determine when the transfer will take place.

The Rx BTC and Tx BTC modules pass data and control information between the Utopia-4 interfaces and internal busses. They also coordinate transfers to optimize packet flow and maintain packet ordering. The Rx BTC assigns receive traffic to threads with either strict round-robin selection, or to the next available thread. Packet transmission follows the strict order of the original packet receive sequence.

The internal Rx and Tx busses provide pathways between the BTCs and BTEs. Each bus can transfer 64 bits of data every cycle. When two port pairs are configured, each BTE is allocated to a specific port via the first write descriptor sent to the BTE. For example, if the first descriptor sent to the BTE in processor 0 specified port 1, processor 0’s BTE would be allocated to port 1 for the remaining descriptors.

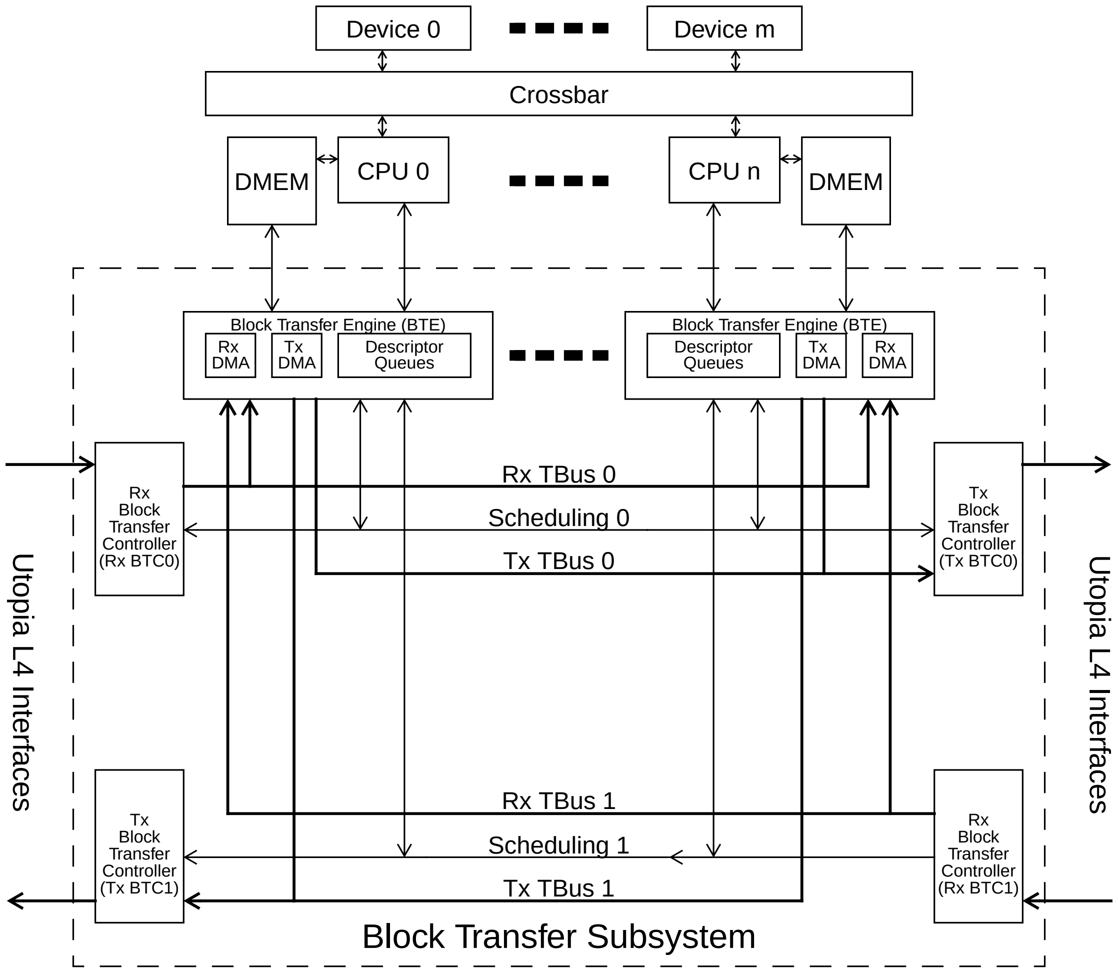

Figure 27: Organization of the Block Transfer Controller

10.2. Block Transfer Buffers and Transfer Descriptors

This section describes the format of the block transfer descriptors and block transfer buffers.

Table 40, Block Transfer Descriptor below shows the format of the descriptors software passes the the BTE using the WD and WD.CSW instructions.

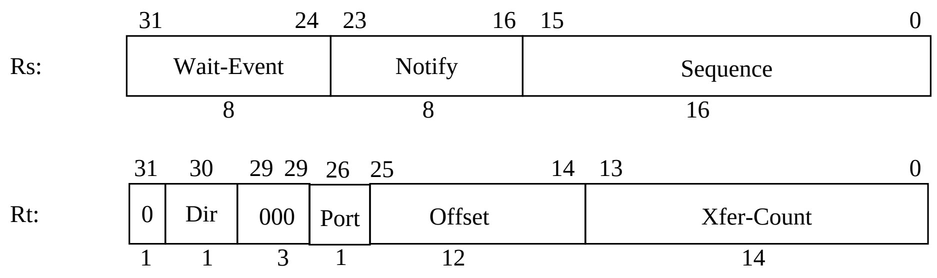

Table 40: Block Transfer Descriptor

Field	Width (Bits)	Description
Wait-Event	8	Wait-Event bits to be set in CXSTATUS when write-descriptor includes a context switch. Rs[24]=1 causes the thread to wait for a receive complete event. Rs[25]= causes the thread to wait for a transmit complete event. This field is valud only with a WD.CSW instruction. For a WD instruction, this field must contain zeroes.
Notify	8	These bits are used to request the BTE to notify the thread of a transfer completion by pulsing the appropriate WAIT-EVENT input to the processor. Rs[16]=1 indicates that the thread wants to be notified after the receive transfer specified by this descriptor is completed. Rs[17]=1 indicates that the thread wants to be notified after the transmit transfer specified by this descriptor is completed.
Sequence	16	Sequencing code used for packet ordering. The BTC inserts a 16-bit sequence code in the reserved field of every received packet transfer buffer. For some modes of operation, software must copy this code to the transmit descriptor to allow the block transfer subsystem to maintain packet ordering. The interpretation of this code is private to the BTE and BTC.
Dir	1	Direction of the transfer. 0 = receive; 1 = transmit
Port	1	ID number of the transmit or receive port to transfer data. All of the threads on a given processor must access a single Rx/Tx port pair at all times.
Offset	12	Offset of the buffer in local DMEM, addressed in 16-byte multiples. The starting location of a buffer must be aligned to a 16-byte boundary. The low order 4 bits of the byte offset are omitted from the Field.
Xfer-Count	14	This value indicates the maximum number of bytes which may be received, or the number of bytes to transmit. It need not be a multiple of 16 bytes. The count does not include the 8 bytes at the beginning of the buffer that are reserved for use by the block transfer subsystem.

Block transfer buffers are regions in a processor's local DMEM that provide storage for receive and transmit packets. The BTE requires the base of each buffer to be aligned to a 16-byte boundary, and the first 8 bytes of the buffer are reserved for use by the BTE. Aside from these restrictions, software has flexibility with regards to the number and size of buffers. Software may maintain an arbitrarily large pool of buffers, within the limits of available local DMEM.

The figure below illustrates the structure of a transfer buffer.

Address								
0	BTC reserved fields (see table below)							
8	oct-1	oct-2	oct-3	oct-4	oct-5	oct-6	oct-7	oct-8
16	oct-9	oct-10	oct-11	oct-12	oct-13	oct-14	oct-15	oct-16
. . .	...	...	...	...	...	...	...	...

BTC Reserved Fields

64-48	47	46	45	44:0
Sequence	Reserved	Abort	Trunc	Reserved

Table 41: BTC Reserved Fields in Transfer Buffer

Field	Use
Sequence	Sequence Number inserted in receive packets by BTE. Ignored for transmit packets.
Abort	A received packet was aborted by the sender.
Trunc	A received packet was truncated by BTE to fit available buffer space.
Reserved	Reserved for future BTE use. Contents undefined for receive packets, ignored for transmit packets.
oct-n	Octet <N> of the receive or transmitted packet.

To make a buffer available for a receive or transmit operation, software (i.e. a thread) creates a descriptor that specifies the buffer location, size and other aspects of the transfer. The thread executes the WD*[.CSW] instruction to enqueue the transfer descriptor in the thread's receive or transmit descriptor queue within the BTE. The descriptor is contained in the general registers specified by the WD* instruction.

10.3.Example Transaction Flow

Here is the flow of an example receive block transfer operation through a NetVortex system:

1. *Enable the Transfer.* A processor initiates a receive block transfer with the WD.CSW instruction (Write Descriptor with Context Switch). Software sets the receive Wait-Event bit, Rs[24], and and the receive Notify bit, Rs[16], to indicate that the thread is waiting for the receive operation to complete and the thread wants to be notified of the completion. The processor performs a context switch to allow the processor to perform work for another thread.

2. *Enqueue the Descriptor.* The processor passes the descriptor to its BTE. In turn, the BTE enqueues the descriptor into the receive descriptor queue that serves the thread. The BTE also informs the Rx BTC that a new buffer is available.

3. *Select the Descriptor.* The BTE analyzes the entries at the heads of its descriptor queues, and selects a descriptor for this transfer. The BTE then passes the descriptor to its Receive DMA (RxDMA) controller for the next transfer.

4. *Schedule the Transfer.* The Rx BTC observes that the BTE is ready to accept data. (This is determined in the background of the current data transfer.)

5. *Transfer the Data.* When the BTE's turn arises, the Rx BTC directs data beats to the BTE, and the BTE's RxDMA controller stores the data into the local DMEM.

6. *Signal the Processor.* After the last beat is transferred, the BTE signals the processor to clear WAIT-EVENT bit 0 of the initiating thread's CXSTATUS register.

7. *Mark Thread Ready.* The processor's thread scheduler marks the applicable thread as ready for execution, if no other events are pending. If another thread is currently executing, a future context switch will activate the thread for which this transfer was completed. If no thread is currently active, the thread is resumed immediately.

8. *Process Packet Data*. The thread is chosen by the thread scheduler and resumes execution. The thread processes the data in the packet and issues a transmit block transfer with the WD.CSW instruction. Software inserts the sequence number from the reserved field of the packet buffer into Rs[15:0]. Bits Rs[25] and Rs[17] are set to indicate that the thread is waiting for the transmit operation to complete and the thread wants to be notified of the completion. The processor performs a context switch to allow the processor to perform work for another thread.

9. *Enqueue the Descriptor.* The processor passes the descriptor to its BTE, which enqueues the descriptor in the transmit descriptor queue that serves the thread that initiated the request.

10. *Descriptor Next to Transmit*. The BTE has a local copy of the currently transmitting sequence number and determines that it has the descriptor for the next transmission. The BTE then passes the descriptor to its Transmit DMA (TxDMA) controller.

11. *Transfer the Data*. When the BTE notices the completion of the current transfer it will begin to transmit data from DMEM to the Tx BTC.

12. *Signal the Processor.* After the last beat is transferred, the BTE signals the processor to clear WAIT-EVENT bit 1 of the initiating thread's CXSTATUS register.

10.4. Detailed Description of Block Transfer Modules

This section provides more a detailed description of the block transfer components.

10.4.1. Block Transfer Engine

A Block Transfer Engine (BTE) is connected directly to each processor, with a dedicated control port to receive descriptors from the processor, a dedicated data port connected to the processors's data RAM and a data port connected to the shared RxBus and TxBus interconnect. The processor connection provides a private pathway from the processor to the engine for writing transfer descriptors with the WD instruction, without using shared system bus bandwidth. Internally, the BTE includes per-thread descriptor queues to manage receive and transmit operations, and dedicated receive and transmit DMA controllers (RxDMA and TxDMA) to pass data between the processor's DMEM and the TBus.

Each thread is supported by dedicated receive and transmit descriptor queues. The BTE has parallel access to the head entry of each queue to determine which descriptor to select for the next receive and transmit operations.

The number of descriptor queue entries is RTL-configurable. The minimum is one entry per descriptor queue, that is, one Rx descriptor and one Tx descriptor. In typical applications, two receive descriptor entries and one transmit descriptor entry are dedicated per thread. Because the BTE allows software to assign the buffer addresses in descriptors, the number of actual packet buffers held in DMEM may be larger than the number of descriptor queue entries. With just three packet buffers, a thread may perform packet computation concurrently with packet receive and transmit operations that take place in the background.

The BTE's DMA controllers can sustain simultaneous receive and transmit operations. The processor's DMEM is a 128-bit wide dual port SRAM, shared by the processor and the BTE. The processor loads or stores only 32 bits of data in any cycle over its port. The 64-bit receive and transmit DMA controllers access the RAM in alternate cycles over their port, writing 128 bits in one cycle and reading 128 bits the next cycle. This allows the RAM to sustain the simultaneous read and write operations that may be presented on 64-bit RxBus and TxBus pathways. The DMA controllers include a register stage to match actual receive and transmit data widths to the 128-bit RAM interfaces.

10.4.2. Rx and Tx Block Transfer Controllers

The Rx and Tx Block Transfer Controllers (Rx BTC and Tx BTC) provide data pathways between the external Utopia-4 Rx/Tx and the internal RxBus and TxBus. The main functions of scheduling are to distribute receive packets among the available pool of receive buffers, and to ensure packets are transmitted in the original receive order. The scheduler enforces the ordering policy by providing sequencing information that is used by the BTEs.

There are two modes of scheduling that may be employed.

- Strict Round-Robin Scheduling.
- Next Available Receive Buffer Scheduling.

Strict Round-Robin Scheduling

Strict round-robin scheduling of incoming packets assigns the first packet to the first processor, the second packet to the second processor, etc. Assignment then wraps around to the first processor, the second processor, etc. Transmit packets are selected in the same order. If the processor that is designated for a receive (or transmit) operation does not have a receive (or transmit) buffer available when required, no data transfer can take place.

Strict round robin scheduling is useful for its simplicity in distributing the packet workload among the processors. It should only be used when a worst case analysis of the traffic flow indicates that receive and transmit operations will not be stalled to wait for a buffer from the next required processor.

Next Available Receive Buffer Scheduling

Next Available receive buffer scheduling assigns an incoming packet, using windowed rotating priority, to a processor that indicates a receive buffer is available. The Rx BTC inserts a sequence code in the reserved field at the start of each received packet. When software is ready to transmit the packet, it inserts the sequence code it obtained from the receive packet into the Sequence field of the transmit descriptor. Each BTE has a local copy of the next sequence code to be transmitted and the Tx BTC informs the BTEs when to increment that code. Each BTE ensures that a transmit descriptor that contains this sequence code is made ready in the BTE's TxDMA controller when the controller is able to accept a new descriptor. The interpretation of the sequence code is hardware implementation specific. Software only needs to copy the sequence code from each receive operation to each transmit operation.

10.4.3. Utopia Level 4 Rx and Tx Interfaces

The Rx BTC and Tx BTC interfaces provide Utopia Level 4 connections to external data paths. A NetVortex system may include one or two Rx/Tx interface pairs. The interfaces support the following features of Utopia Level 4:

- 415 MHz operation.
- 32-bit wide data path.
- Channel 0 Provisioning.
- Concatenated (non-channelized) data streams.
- Packet and ATM cell transfer modes.
- Flow control signalling.

All of Rx/Tx interface signals are differential, so each port requires a total of 68 device pins. They are summarized in the tables below.

Table 42: Receive Port Signals

signal	direction	description
RXU4_CLKI	input	receive reference clock
RXU4_CTLI	input	receive control flag
RXU4_DATAI[31:0]	input	receive data

Table 43: Transmit Port Signals

signal	direction	description
TXU4_CLKO	output	transmit reference clock
TXU4_CTLO	output	transmit control flag
TXU4_DATAO[31:0]	output	transmit data

Appendix A. NetVortex Lconfig Forms

A.1. Introduction

The general construct of the multi-processor form is a block. The block can refer to a configuration for use later in the form, or it can be used as a declaration inside another block. The structure of a block:

```
BLOCK_TYPE BLOCK_NAME {...};
```

The *BLOCK_TYPE* parameter is one of a handful of Lconfig keywords. It can describe a processor, a crossbar device, or a system. Here is a list of valid block types and their descriptions:

Table 44: Lconfig Block Types

Block Types	Description	Section
PPU	Packet Processing Unit	Section A.2, Packet Processor Unit
NVX_4PROC	NetVortex 4 processor tile with a Level 1 crossbar	Section A.3, Four Processor Tile with Level 1 Crossbar
MEMORY_MAP	Memory-Mapped crossbar device	Section A.4, Memory-Mapped Cross-bar Device
WRITE_DESC	Write-Descriptor crossbar device	Section A.5, Write-Descriptor Cross-bar Device
TASER	Test and Set Engine crossbar device	Section A.6, Test and Set Engine Crossbar Device
CROSSBAR	NetVortex system including the Level 2 crossbar	Section A.7, NetVortex System with Level 2 Crossbar

The *BLOCK_NAME* is user defined, and may be used by Lconfig to help generate files for that configuration. For instance, the *BLOCK_NAME* parameter in the PPU block is used as a prefix for the processor rtl files.

The block itself contains form options, verilog symbol declarations, and declarations of other blocks. The multi-processor NetVortex form is intended to model the actual system hierarchy as closely as possible.

A.2. Packet Processor Unit

```
PPU PPU_CONFIG_NAME {
  FORM_OPTION = VALUE;
  ...
  SYMBOL `define VERILOG_SYMBOL VALUE
  ...
};
```

The PPU block does not contain sub-block instances or declarations and it uses the same processor form options as LX8000 configurations.

A.3. Four Processor Tile with Level 1 Crossbar

```
NVX_4PROC NVX_4PROC_CONFIG_NAME
{
      PPU PPU_DECL_NAME1 = PPU_CONFIG_NAME;
```

```
        ...
    };
```

The NVX_4PROC block contains the instances of the packet processors. If a PPU configuration is declared above this block, it may be referenced inside. Four processors are required in the block.

A.4. Memory-Mapped Crossbar Device

```
MEMORY_MAP BLOCK_NAME {
  FORM_OPTION = VALUE;
  ...
};
```

The MEMORY_MAP block describes a memory-mapped device. This block specifies a base address and an address mask that declares the range of addresses defining a memory device. If the device specifies a 128 MB range, for instance, then the base address must be on a 128 MB boundary. Memory-maps must not overlap with each other, including the write-descriptor memory-map. Here is a summary of address masks and the memory window size created:

Table 45: Memory Mapped Device Address Masks

ADDR_MASK	SIZE
12'h000	4 GB
12'h800	2 GB
12'hC00	1 GB
12'hE00	512 MB
12'hF00	256 MB
12'hF80	128 MB
12'hFC0	64 MB
12'hFE0	32 MB
12'hFF0	16 MB
12'hFF8	8 MB
12'hFFC	4 MB
12'hFFE	2 MB
12'hFFF	1 MB

A.5. Write-Descriptor Crossbar Device

```
WRITE_DESC BLOCK_NAME {
  FORM_OPTION = VALUE;
  ...
};
```

The WRITE_DESC block describes a write-descriptor device and will respond to write-descriptor instructions. Each write-descriptor device responds to one or more DEVICE_ID bits. However, any given DEVICE_ID bit must map to one and only one write-descriptor device.

A.6. Test and Set Engine Crossbar Device

```
TASER BLOCK_NAME {
  FORM_OPTION = VALUE;
  ...
};
```

The TASER block describes a Test and Set Engine device. This device is a special memory-mapped device in that it allows access to semaphores. It has the same basic form options as a memory-mapped device, with one additional option used to configure the number of semaphores in the Test and Set Engine. Only one TASER device may be used in a NetVortex system.

A.7. NetVortex System with Level 2 Crossbar

```
CROSSBAR {
  FORM_OPTION = VALUE;
  ...
  NVX_4PROC NVX_4PROC_DECL_NAME1 = NVX_4PROC_CONFIG_NAME;
  ...
  MEMORY_MAP MM_DECL_NAME {
    FORM_OPTION = VALUE;
    ...
  };
  ...
  WRITE_DESC WD_DECL_NAME {
    FORM_OPTION = VALUE;
    ...
  };
  ...
  TASER TASER_DECL_NAME {
    FORM_OPTION = VALUE;
    ...
  };
};
```

The CROSSBAR block instances an entire NetVortex system. The processor tiles are declared, the device ports are configured, and the related crossbar form options are set. This block does not need a *BLOCK_NAME* since it may be instanced only once. The order in which crossbar devices are declared is the order they will be inserted into the crossbar ports. The first device will be connected to port 0, the next device to port 1, etc. There must be at least one crossbar device, up to a maximum of six. Only one TASER device may be used in the system. Memory-mapped devices may not have overlapped ranges, and write-descriptor devices may not share the same write-descriptor bits.

A.8. General Form Notes

When using a particular configuration more than once, for example, when declaring a 4 processor tile, if all 4 processors are identical, instance each processor with the same configuration:

```
NVX_4PROC PPU4TILE {
  PPU pA = PACKETPU;
  PPU pB = PACKETPU;
  PPU pC = PACKETPU;
  PPU pD = PACKETPU;
};
```

This guarantees that Lconfig will use one type of packet processor (PACKETPU in this case) for all processors in the 4 processor tile. If the processors are explicitly declared inside the block, Lconfig will parse each processor independently, even if they are exactly the same, and each configuration will be synthesized separately. Therefore, when using a configuration more than once, describe the processor with a configuration block, and instance it in the processor tile as shown above.

All crossbar devices must be uniquely addressable. To avoid accidently instancing a crossbar device configuration twice inside the crossbar block, explicitly declare each crossbar device:

```
CROSSBAR {
  ...
    MEMORY_MAP DEVICE0 {
    FORM_OPTION = VALUE;
    ...
  };
  ...
};
```

A.9. Example NetVortex Form

Below is an example NetVortex form. This form defines a four processor system, a synchronous crossbar, and three devices (a memory-mapped device, a write-descriptor device, and a Test and Set Engine). The four processors are identical

```
PPU NVP {
  PRODUCT               =  "LX8000";
  PRODUCT_TYPE          =  "RTL";
  TECHNOLOGY            =  "CUSTOM";
  CUSTOM_FILES          =  "YES";
  TESTBED_ENV           =  "CHIP";
  RESET_TYPE            =  "ASYNCHRONOUS";
  RESET_DIST            =  "GLOBAL";
  SLEEP                 =  "NO";
  RESET_BUFFERS         =  "LX2";
  CLOCK_BUFFERS         =  "LX2";
  RAM_CLOCK_BUFFERS     =  "NO";
  COP1                  =  "NONE";
  COP2                  =  "EXPORT";
  COP3                  =  "NONE";
  CE0                   =  "CE_HL";
  CE1                   =  "NONE";
  MEM_LINE_ORDER        =  "SEQUENTIAL";
  MEM_FIRST_WORD        =  "ZERO";
  MEM_GRANULARITY       =  "BYTE";
  SYSTEM_INTERFACE      =  "LBUS";
  WDESC_ADDR            =  "12'hFF6";
  LBC_WBUF              =  "4";
  LBC_RBUF              =  "4";
  LBC_RDBYPASS          =  "YES";
  LBC_SYNC_MODE         =  "SYNCHRONOUS";
  LINE_SIZE             =  "4";
  ICACHE                =  "NONE";
  DCACHE                =  "NONE";
  IMEM                  =  RANGE(0x4040_0000,0x4040_1fff);
  DMEM                  =  RANGE(0x4051_0000,0x4051_1fff);
  DMEM_WIDTH            =  "128";
  LMI_RANGE_SOURCE      =  "HARDWIRED";
  LMI_RAM_ARB           =  "NO";
  JTAG                  =  "EXPORT_EXTENDED";
  EJTAG                 =  "YES";
  EJTAG_INST_BREAK      =  "2";
  EJTAG_DATA_BREAK      =  "2";
  PC_TRACE              =  "NO";
  EJTAG_DCLK_N          =  "3";
  EJTAG_TPC_M           =  "8";
  EJTAG_XV_BITS         =  "4";
  EJTAG_PC_ISABIT       =  "NO";
  SCAN_INSERT           =  "NO";
  SCAN_MIX_CLOCKS       =  "NO";
  SCAN_NUM_CHAINS       =  "4";
  SCAN_SCL              =  "NO";
  RAM_BIST_MUX          =  "NO";
  LEXOP2_OPCODE         =  "LX2OP";
  LEXOP2_DISABLE        =  "NO";
  THREAD_SCHEDULER      =  "INTERNAL";
  CONTEXTS              =  "8";
  JTAG_TRST_IS_TPC      =  "NO";
};


NVX_4PROC PPU4 {
  PPU pA                =  NVP;
  PPU pB                =  NVP;
  PPU pC                =  NVP;
  PPU pD                =  NVP;
};


CROSSBAR {
  CROSSBAR_SYNC_MODE       =           "SYNCHRONOUS";

  NVX_4PROC XB0         =  PPU4;
  // uncomment this line to make it 8 processors
  // NVX_4PROC XB1      =  PPU4;

  MEMORY_MAP MEMORY { // This device is on port 0
    READ_DEVICE            =           "YES";
    ADDR_BASE              =           "12'h000";
    ADDR_MASK              =           "12'h800";
    REQ_QSIZE              =           "8";
    READ_QSIZE             =           "8";
  };
  WRITE_DESC CAM { // This device is on port 1
    READ_DEVICE           =          "YES";
    DEVICE_ID             =          "32'hFFFFFFFF";
    REQ_QSIZE             =          "8";
    READ_QSIZE            =          "8";
  };

  TASER TESTANDSET { // This device is on port 2
      READ_DEVICE      =  "YES";
      ADDR_BASE        =  "12'hFF8";
      ADDR_MASK        =  "12'hFFF";
      REQ_QSIZE        =  "8";
      READ_QSIZE       =  "8";
      SEMAPHORES       =  "32";
  };
};
```

This form describes a 4 processor system, synchronous crossbar interface, with a 2GB memory mapped device, a write-descriptor device, and a Test and Set Engine device. The device port for the memory-mapped device has an 8 entry queue for the device request path, and an 8 entry queue for the read return path. Any DEVICE_ID will select the write-descriptor device on port 1 of the crossbar. This port also has an 8 entry queue for the device request path and an 8 entry queue for the read return path. The Test and Set Engine device is on crossbar port 2, and has a 1 MB memory-mapped window starting at FF80_0000h. It is configured for 32 addressable semaphores, and the crossbar has 8 entries for the request and read return paths for the Test and Set Engine's port.

By declaring the "PPU4" construct, it is easy to reuse the tile description for 8, 12, or 16 processor configurations.

A.10. 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
```

A.11. Configuration Options for Memory-Mapped Devices

```
READ_DEVICE       -- Does the device return read data
ADDR_BASE         -- Base Address for Memory mapped range (31:20)
ADDR_MASK         -- Address Mask used to specify the size of the map
                     (31:20)
REQ_QSIZE         -- XBar Queue size for device request path
READ_QSIZE        -- XBar Queue size for device read path
```

A.12. Configuration Options for the Test & Set Engine

```
READ_DEVICE       -- Does the device return read data
ADDR_BASE         -- Base Address for Memory mapped range (31:20)
ADDR_MASK         -- Address Mask used to specify the size of the map
                     (31:20)
REQ_QSIZE         -- XBar Queue size for device request path
READ_QSIZE        -- XBar Queue size for device read path
SEMAPHORES        -- No. of semaphores for the test and set engine
```

A.13. Configuration Options for Write-Descriptor Devices

```
READ_DEVICE       -- Does the device return read data
DEVICE_ID         -- 32-bit Write Descriptor Device ID
REQ_QSIZE         -- XBar Queue size for device request path
READ_QSIZE        -- XBar Queue size for device read path
```

A.14. Configuration Options for the Crossbar

```
CROSSBAR_SYNC_MODE-- Sync/Async Crossbar Interconnect
```

Appendix B. NetVortex Port Descriptions

The table below shows the port connections for the NetVortex top level module, known as nvx3. Ports that are replicated for multiple processors include <n> in the name, where <n> = 0, 1, up to 15. Likewise, device ports include <m> in the port name, where <m> = 0, 1, up to 5.

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).

Table 46: NetVortex Top Level Port Summary

Port Name	I/O	Timing	Description
Clocking, Reset, Interrupts, and Control			
SYSCLK	input	(3)	Processor clock.
BUSCLK	input	(3)	Asynchronous crossbar clock.
ResetN	input	(4)	Warm reset, e.g. from a button or higher level controller.
CResetN	input	(4)	Cold reset, from power on condition.
p<n>_RESET_D1_R_N_O	output	30%	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.
XB_RESET_D1_R_N	input	(4)	SYSCLK domain reset input for cross-bar, connect to p0_RESET_D1_R_N_O output.
XB_RESET_D1_BR_N	input	(4)	BUSCLK domain reset input for cross-bar, connect to p0_RESET_D1_R_N_O output syn-chronized into BUSCLK domain.
p<n>_INTREQ_N[15:2]	input	(4)	Interrupt requests, active low.
EXT_HALT_P	input	50%	Drive to one stalls processor next cycle.
Configuration			
CFG_MEMSEQUENTIAL	input	(4)	Strap to one if line reads return words in sequential order, zero if interleave order. Tie to 1.

Port Name	I/O	Timing	Description
CFG_MEMZEROFIRST	input	(4)	Strap to one if line reads return word zero first, zero if desired word first. Tie to 1.
CFG_LBCWBDISABLE	input	(4)	Strap to one to disable read bypass of LBC write buffer, zero to allow read bypass. Tie to 1.
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 configura-tion.
CFG_EJTBIT0M16	input	(4)	Strap with EJTAG PC bit 0 in TPC con-figuration.
CFG_EJDIS	input	(4)	Must be strapped to zero.
CFG_DWDISW	input	(4)	Strap to one to disable processor DMEM writes. Must be zero for NetVor-tex.
Test and Debug			
JTAG_TDO_NR	output	50%, (1)	Test data out.
JTAG_TDI	input	50%, (1)	Test data in.
JTAG_TMS	input	60%, (1)	Test mode select.
JTAG_CLOCK	input	(3)	Test mode select.
JTAG_TRST_N	input	(4)	Tap controller reset.
p<n>_EJC_ECRPROBEEN_R	output	30%	One indicates EJTAG probe is active.
p<n>_RBC_SEL[7:0]	input	(4)	RAM BIST RAM select code: 10000000 - instruction MEM 01000000 - not used 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 Note for NetVortex, the DMEM is not accessible via the RAM BIST path.
p<n>_RBC_WE[<k>:0]	input	(4)	RAM BIST write enable, where <k> is 1 for word write granularity, 7 for byte write granularity.
p<n>_RBC_RE	input	(4)	RAM BIST read enable.
p<n>_RBC_CS	input	(4)	RAM BIST select.
p<n>_RBC_ADDR[15:0]	input	(4)	RAM BIST address.
p<n>_RBC_DATAWR[63:0]	input	(4)	RAM BIST write data.

Port Name	I/O	Timing	Description
p<n>_RBM_DATARD[63:0]	output	(4)	RAM BIST read data.
Data RAM DMA Access			
p<n>_DMADW_RCLK	input	(3)	Data RAM DMA clock (optional)
p<n>_DMADW_DATAINDEX[17:4]	input	(5)	Data RAM DMA address.
p<n>_DMADW_DATARD[127:0]	output	(5)	Data RAM DMA read data.
p<n>_DMADW_DATAWR[`127:0]	input	(5)	Data RAM DMA write data.
p<n>_DMADW_DATACS	input	(5)	Data RAM DMA chip select.
p<n>_DMADW_DATACSN	input	(5)	Data RAM DMA chip select, active low.
p<n>_DMADW_DATARE	input	(5)	Data RAM DMA read enable.
p<n>_DMADW_DATAREN	input	(5)	Data RAM DMA read enable, active low.
p<n>_DMADW_DATAWE[<k>:0]	input	(5)	Data RAM DMA write enable, where <k> is 3 for word write granularity, 15 for byte write granularity.
p<n>_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.
Coprocessor Interface			
p<n>_C2condin	input	80%	Cop branch flag.
p<n>_C2rd_addr[4:0]	output	50%	Cop read address.
p<n>_C2rhold	output	45%	Cop hold condition, one stalls coprocessor.
p<n>_C2rd_gen	output	50%	Cop general register read command.
p<n>_C2rd_con	output	50%	Cop control register read command.
p<n>_C2rd_data[31:0]	input	80%	Cop read data.
p<n>_C2wr_addr[4:0]	output	20%	Cop write address.
p<n>_C2wr_gen	output	20%	Cop general register write command.
p<n>_C2wr_con	output	20%	Cop control write address command.
p<n>_C2wr_data[31:0]	output	30%	Cop write data.
p<n>_C2invld_M	output	60%	Cop invalid instruction flag, one indicates invalid instruction in M stage.
p<n>_C2xcpn_M	output	60%	Cop exception flag, one indicates exception in M stage.
p<n>_C2rd_cntx[2:0]	output	40%	Cop read context number.
p<n>_C2wr_cntx[2:0]	output	30%	Cop write context number.
Event Control and Thread Observation			

Port Name	I/O	Timing	Description
p<n>_EXT_CLEARWTEVNT_R [<n>*8-1:0]	input	30%	External hardware clear wait event flags. <n> is the number of contents.
p<n>_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.
p<n>_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.
p<n>_EXT_NXTCNTX_P_R[2:0]	input	30%	External Scheduler Next Context.
p<n>_EXT_NEXTCNTXRDY_P_R	input	30%	External Scheduler Next Context is ready.
p<n>_CX_STUSTHPRIO_R[<n>*3-1:0]	output	30%	Thread priority status.
Error Signalling			
XB_ErrBadAddr<n>	output	(2), 20%	A bit pulsed high identifies a processor that passed a bad address to crossbar, where <n> is the processor.
XB_ErrInvldRd<n>	output	(2), 20%	A bit pulsed high identifies a processor that passed a bad read request to crossbar, where <n> is the processor.
XB_ErrThread<n>[3:0]	output	(2), 20%	Identifies error causing thread that signals an error via XB_ErrBadAddr or XB_ErrInvldRd. <n> is the processor.
Crossbar Device Interface			
DevReqRdy<m>	output	(2), 20%	Asserted when a new request is present.
ReqAddr<m>[31:0]	output	(2), 20%	Request address.
ReqCmd<m>[3:0]	output	(2), 20%	Request command.
ReqSize<m>[2:0]	output	(2), 20%	Request size.
ReqGTid<m>[15:0]	output	(2), 20%	Request global thread ID.
ReqData<m>[31:0]	output	(2), 20%	Write data (for memory mapped devices).
ReqDevID<m>[4:0]	output	(2), 20%	Write descriptor device ID (for write descriptor devices).
ReqData<m>[63:0]	output	(2), 20%	Write descriptor write data (for write descriptor devices).
DevBusy<m>	input	(2), 30%	Device will not accept current request when asserted.
RdDequeue<m>	output	(2), 40%	Asserted by crossbar when it accepts read data.
RdDataRdy<m>	input	(2), 30%	Device's split read data is ready.

Port Name	I/O	Timing	Description
SplitRdData<m>[63:0]	input	(2), 30%	Split read data.
RdGTid<m>[15:0]	input	(2), 30%	Global thread ID associated with split read data.
RdCmd<m>[1:0]	input	(2), 30%	Read data command.
RdSize<m>[1:0]	input	(2), 30%	Read data size.
Device Management Interface			
DMI_ProcNum[7:0]	input	(2), 20%	Processor number assigned to DMI.
DMI_DevReqs[m-1:0]	input	(2), 20%	DMI request lines, to devices. <m> is the number of devices.
DMI_Gnt	output	(2), 20%	DMI grant lines, from devices.
DMI_Cmd[3:0]	input	(2), 20%	DMI command, to device.
DMI_Size[2:0]	input	(2), 20%	DMI data size, to device.
DMI_Addr[31:0]	input	(2), 20%	DMI address, to device.
DMI_Data[63:0]	input	(2), 20%	DMI write data, to device.
DMI_ThreadID[3:0]	input	(2), 20%	DMI thread ID, to device.
DMI_DevID[4:0]	input	(2), 20%	DMI device address, to device.
DMI_RdValid	output	(2), 20%	DMI split read data valid, from device.
DMI_RdBusy	input	(2), 30%	DMI read path busy, to device.
DMI_RdData[63:0]	output	(2), 20%	DMI split read data, from device.
DMI_RdThreadID[3:0]	output	(2), 20%	DMI split read thread ID, from device.
DMI_RdSize[1:0]	output	(2), 20%	DMI split read data size, from device.
DMI_RdCmd[1:0]	output	(2), 20%	DMI split read data command, from device.

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 47: LX8000 Single-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.

Port Name	I/O	Timing	Description
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			
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).

Port Name	I/O	Timing	Description
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.)
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

Port Name	I/O	Timing	Description
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.
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.

Port Name	I/O	Timing	Description
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.
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.

Port Name	I/O	Timing	Description
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.
C<z>rhold	output	45%	Cop hold condition, one stalls copro-cessor.
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 indi-cates 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

Port Name	I/O	Timing	Description
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.
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 48: 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 49: 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
```