LX5280 Data Sheet

Lexra, Inc.

Release 1.9

April 30, 2001

Lexra Proprietary and Confidential

LX5280 Data Sheet Revision 1.1, for RTL Release 1.9.

Table of Contents

List of Tables

List of Figures

1. LX5280 Product Overview

1.1. Introduction

This data sheet describes the LX5280, a high-performance RISC-DSP, developed for Intellectual Property (IP) licensing. However, the LX5280 is not just a highly specialized DSP architecture, but also a carefully engineered extension to the MIPS ISA. As a result, system functions and computationally intensive DSP algorithms can be integrated on a single, low-cost subsystem. Key applications include data communication products such as network protocol processors, cable and xDSL modems, Voice over Packetized Network gateways, and set-top boxes, as well as disk controllers.

As DSP-intensive applications have gained commercial importance, there has been an increasing recognition of the benefit of implementing DSP functions on the CPU. A CPU is usually required for memory management, user interface and control software; CPUs also have excellent third-party software tool support. However, software implementations of DSP algorithms such as the FIR Filter or Discrete Cosine Transform (DCT) typically suffer by an order of magnitude in performance compared to specialized DSPs. The problem is compounded by the difficulty of deterministically allocating real-time in sophisticated CPUs. Vendors have addressed these problems by offering *DSP Coprocessors* which have separate instruction sets, separate instruction stores and execution units; and *DSP Accelerators*, which share the same I-stream with the CPU but have separate DSP execution units. Each of these approaches imposes a substantial burden on the CPU in managing the DSP functions. The LX5280, on the other hand, tightly integrates its DSP extensions into the MIPS ISA. As a result, a wide variety of third-party tools are available and the LX5280 programmer can switch seamlessly from RISC code to DSP code.

The LX5280 adds to the MIPS-I instruction set a collection of DSP-oriented instructions called the Radiax™ instruction set. The Radiax instruction set adds dual 16-bit multiply and multiply-accumulate operations including DSP modes such as saturation, rounding and fractional arithmetic. It includes DSP addressing modes such as post-modified address pointers, circular buffers and zero-overhead loops. It also includes dual 16-bit SIMD ALU operations and data alignment operations for applications where 16 bits of data is sufficient.

The LX5280 pipeline is a dual-issue, six-stage architecture. Pipe A is the *Load/Store Pipe* and includes data memory access and all MIPS instructions except multiply and divide operations, while Pipe B is the *Multiply-Accumulate Pipe* and includes a MAC and ALU, each with dual 16-bit operations. DSP algorithms will typically use Pipe A to load a pair of operands into a general register while executing Dual MAC operations in Pipe B on earlier data. Decoupling register loads from the MAC allows loop unrolling and takes effective advantage of the 32 general registers for temporary storage. As compared to memory-based operands which are common to specialized DSP instruction sets, dual-issue allows the LX5280 to achieve the memory bandwidth required by DSP within a RISC architecture. In addition, the LX5280 introduces twinword load and store functions, allowing 64-bits of data to be moved between the local cache and register file in a single cycle. This provides sufficient data-movement for data-hungry DSP inner loops.

Features introduced in Lexra's RISC product line to support System-on-Chip (SoC) design, including customer-defined Coprocessors and customer extensions to the MIPS ISA, are standard in the LX5280. Configuration options include Extended-JTAG (EJTAG) support for debug and In-Circuit Emulation (ICE). Lexra's products include the same memory management stub (SMMU) as the LX4189.

Because the LX5280 executes the MIPS instruction set, a wide variety of third-party software tools are available including compilers, operating systems, debuggers and in-circuit emulators. The assembler extensions and a cycle accurate Instruction Set Simulator (ISS) are developed by Lexra. Programmers can use "off-the-shelf" C Compilers for initial coding; then replace performance-critical loops with optimized assembler code.

Third parties provide C compiler support for the new DSP instructions and will supply DSP macro libraries

and application packages. Compiler support is provided by the GreenHills MULTI IDE package. A DSP library of functions such as filters and transforms will be available from Lexra.

Memory sizes and peripherals can be tailored by the licensee to system requirements, avoiding excess silicon cost and power dissipation. It is expected that the LX5280 will deliver exceptional price/performance for numerous consumer products and that multiple LX5280 subsystems on a single die can cost-effectively implement high-performance next generation telecommunications systems.

Key Features

- **Complete Processor Subsystem**
  - Executes MIPS I ISA (except unaligned loads, stores).
  - Extensive third-party tool support.
  - Dual instruction issue.
  - High-performance 6-stage pipeline.
  - Local instruction memory and/or cache, configurable sizes.
  - Local data memory and/or cache, configurable sizes.
  - Memory interface logic included.
  - System bus controller.
  - Optional customer-defined coprocessors.
  - Optional customer-defined instruction extensions.
  - Supports EJTAG Draft 2.0 for debugging.
- **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.
- **Easy ASIC Design**
  - Single phase clocking.
  - Fully synchronous design.
  - Easy to interface system bus protocol.
  - Supports popular EDA tools.
- **Executes Lexra's Radiax™ Instruction Set**
  - SIMD operations.
  - Zero-overhead loop.
  - Multiply-accumulate instructions.
  - Vector and circular buffer addressing modes.
- **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.

1.2. LX5280 Processor Overview

The LX5280 is a RISC-DSP processor that executes the MIPS-I instruction set[1] along with Lexra's Radiax™ DSP 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 submicron process technology, as well as design methodology advances.

The figure below shows the structure of the LX5280 processor.

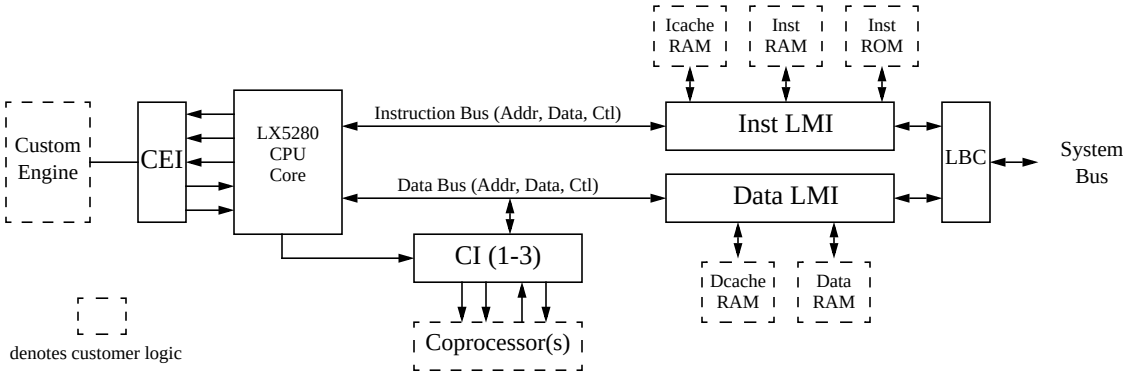


Figure 1: LX5280 Processor Overview

MIPS ISA Execution. The LX5280 supports the MIPS I programming model. Two source operands can be supplied and one destination update performed per cycle. The second operand is either a register or 16-bit immediate. The instruction set includes a wide selection of ALU operations executed by the RALU, Lexra's proprietary register based ALU. The RALU also generates memory addresses for 8-bit, 16-bit, and 32-bit register loads from (stores to) memory by adding a register base to an immediate offset. An extension to the MIPS ISA allows a pair of 32-bit registers to be loaded from (stored to) memory. Branches are based on comparisons between registers, rather than flags, and are therefore easy to relocate. Optional links following jump or branch instructions assist with subroutine programming.

The MIPS unaligned load and store instructions are not supported, because they represent poor price/performance trade-off for embedded applications.

Pipeline. LX5280 instructions are executed by a six-stage pipeline that has been designed so that all transactions internal to the LX5280, as well as at the interfaces, occur on the positive edge of the processor clock. Two-phase clocks are not used.

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 LX5280 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

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

stalls.

LX5280 provides excellent price/performance and time-to-market. There are two main approaches which Lexra has taken to achieve this:

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

Section 1.3 describes the building blocks, and Section 1.4 describes the deliverable models.

1.3. System Level Building Blocks

The LX5280 processor is designed to easily fit into different target applications. It provides the following building blocks.

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

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

1.3.1. SMMU

The LX5280 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.

1.3.2. Local Memory Interface

The LX5280's Harvard Architecture provides Local Memory Interfaces (LMIs) that support instruction memory and data memory. Synchronous memory interfaces are employed for all memory blocks. The LMI block is designed to easily interface with standard memory blocks provided by ASIC vendors or by third-party library vendors.

The LMIs provide a two-way set associative instruction cache interface, and a direct-mapped write-through data cache interface. The tag compare logic as well as a cache replacement algorithm are provided as part of the LMI. One of the instruction cache sets may be locked down as un-swappable local memory. Local instruction and data memories can also be mapped to fixed regions of the physical address space, and include non-volatile memory (such as ROM, flash, or EPROM).

1.3.3. Coprocessor Interface

Lexra supplies an optional Coprocessor Interface (CI) for applications requiring this functionality. Up to three CIs may be implemented in one design. The Coprocessor Interface "eavesdrops" on the Instruction bus. If a Coprocessor load (LWCz) or "move to" (MTCz, CTCz) is decoded, data is passed over the Data Bus into a CI register, then supplied to the designer-defined Coprocessor. Similarly, if a Coprocessor store (SWCz) or "move from" (MFCz, CFCz) is decoded, data is obtained from the Coprocessor and loaded into a CI register, then transferred onto the Data Bus in the following cycle. The design interface includes a data bus, five-bit address, and independent read and write selects for Coprocessor registers and control registers. The LX5280 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.

1.3.4. Custom Engine Interface

The LX5280 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 LX5280. To support multi-cycle operations, a stall input is included in the interface.

1.3.5. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the interface between the LX5280 and the outside world, which includes DRAM and various peripherals. It is a non-multiplexed, non-pipelined, and non-parity checked bus to provide the easiest bus protocol for design integration. On the processor side, the LBC provides a write-buffer of configurable depth to support the write-through cache, as well as the control for byte and half-word transfers. On the peripheral side, the LBC is designed to easily interface to industry standard bus protocols, such as PCI, USB, and FireWire.

The LBC can run at any speed from 33 MHz, up to the speed of the LX5280 processor core in both the RTL core and SmoothCore.

1.3.6. Building Block Integration

The LX5280 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 LX5280 core.

1.4. RTL Core & SmoothCore

Lexra delivers LX5280 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 LX5280.

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, the LX5280 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.

1.5. EDA Tool Support

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

Table 1: EDA Tool Support

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

2. LX5280 Architecture

2.1. Motivation

The LX5280 issues dual 32-bit instructions to two distinct 6-stage execution pipelines. *Superscalar* architectures have been widely deployed in RISC CPUs where increased performance is obtained at the cost of significantly increased area. Although superscalar issue significantly increases the area of the LX5280 processor Core, performance analysis at Lexra demonstrates benefits on key DSP algorithms well beyond that which is obtained in typical CPU benchmarks.

Sustaining peak computational performance in DSP algorithms typically requires at least one operand from memory per instruction cycle. DSPs have traditionally implemented specialized instruction sets that support memory-based operands. Single-issue RISC architectures operate on register-based operands and thus degrade performance by a factor of two in order to pre-load the operand into the register file. Grafting memory-based operands onto a RISC architecture is inconsistent with both the RISC pipeline and ISA. Dual-issue superscalar design, on the other hand, allows operands to be loaded from memory by one instruction while the Multiple Accumulate data path (MAC) operates on register-based data loaded earlier from memory. The RISC data path is 32-bits, but few DSP algorithms require more than 16-bits of precision. Thus two values can be fetched simultaneously from memory. Simultaneous dual 16-bit ALU and MAC operations further improve the LX5280 DSP performance.

Compared to specialized 32-bit (or even 16-bit) DSP instructions which allow memory reference, the superscalar approach will have lesser code density, but only within the DSP loop, or kernel. These kernels are typically small sections of code which are executed many times. Thus the overall degradation of code density is minimal and can be offset by use of MIPS16 code compression in “outer loop” code which is not performance critical.

2.2. Hardware Architecture

2.2.1. Module Partitioning

The LX5280 processor core includes two major blocks: the RALU (register file and ALU) and the CP0 (Control Processor). The RALU performs ALU operations and generates data addresses while CP0 includes instruction address sequencing, exception processing, and product specific mode control. The RALU and CP0 are loosely-coupled and include their own independent instruction decoders.

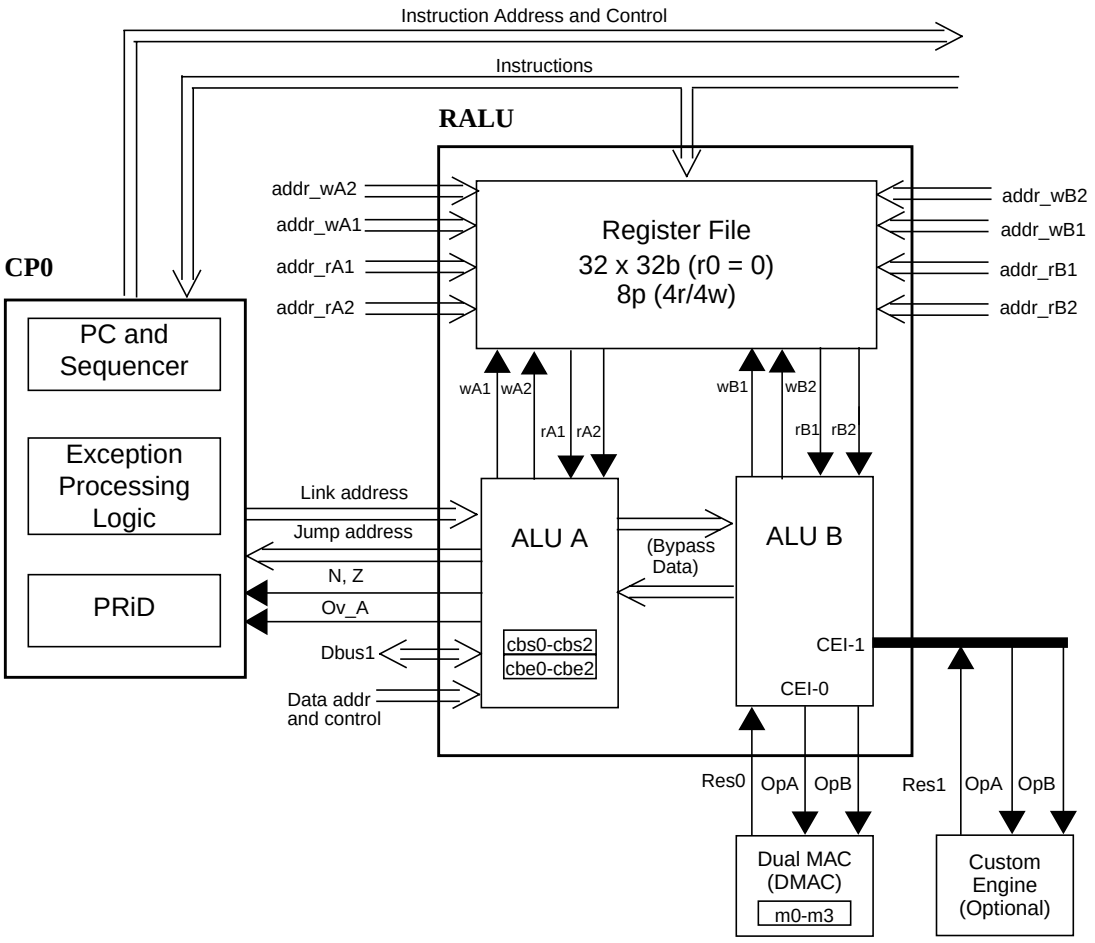


Figure 2: Superscalar Processor Core Module Partitioning

2.2.2. Six Stage Pipeline

The LX5280 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 LX5280 I-Cache and IRAM can fetch two 32-bit instructions I0_I, I1_I simultaneously. Following the superscalar instruction buffer and issue logic, described below, the instructions are issued to Pipe B and Pipe A as appropriate. To avoid degrading operating frequency, the superscalar issue logic operates during the Decode stage (D-stage) of the pipeline. Support for fully synchronous memories in the LX5280 has the added benefit of isolating the processor logic from the customer-supplied memories in the instruction cache, thus facilitating integration of the LX5280 into SoC designs.

As a result of the D-Stage, a two cycle penalty is incurred on branch prediction failure vs. the one-cycle penalty in the LX4180 five stage pipeline. However, the LX5280's zero-overhead loop hardware and

conditional move instructions can be used to avoid any wasted cycles in the control of real-time critical loops.

2.3. Dual Issue

2.3.1. Instruction Fetch

Two instructions are fetched during each instruction cache access. In the event of a cache miss, the processor will be stalled until the cache line containing the requested instructions is retrieved. In the event that only one instruction of a fetched pair can issue, the fetch will be stalled until the second instruction is issued to the pipeline.

Instruction fetches always occur on an aligned 64-bit address boundary. In the event of a branch to an odd 32-bit address in the 64-bit boundary, both instructions in the 64-bit window will be fetched, but only the second (odd) instruction will issue to the pipeline. The first, or even, instruction will be ignored.

2.3.2. Instruction Analysis and Select Logic

The Instruction Analysis and Select Logic is located in the D-stage of the pipeline. During this stage, the processor analyzes both instructions in a fetched pair and determines which pipeline can execute the instructions. For example, if the first instruction in the pair, I0, is an ADD, and the second instruction I1is a MAC, the processor will determine that I0 can be executed by either Pipe A or Pipe B while I1 can be executed by Pipe B. The Instruction Select Logic will then issue I0 to pipe A and I1 to pipe B, since only pipe B can execute the MAC instruction.

If both instructions of the fetched pair can only be issued to one pipeline (for example, a pair of MAC instructions, which can only issue to Pipe B), the two instructions will be issued serially. The instruction fetch will be stalled by one cycle until the second instruction has been issued to the pipeline.

If the result of the first instruction, I0, is used by the second instruction, I1, only one of the two instructions will issue. The second instruction, I1, will issue in the next cycle, and the instruction fetch will be stalled for one cycle until I1 has been issued.

2.3.3. MIPS16

The MIPS16_N signal indicates whether or not MIPS16 code compression has been enabled. If so, each 32-bit fetch is interpreted as a pair of 16-bit instructions encoded according to the MIPS16 Specification. MIPS16 instructions are not dual-issued, but always issued to Pipe A. It is expected that MIPS16 code compression is enabled for “outer loop” code where code density is more important than performance. The critical Register File read addresses for MIPS16 are resolved during the D-stage so that register file access for MIPS16 instructions, as for 32-bit MIPS instructions, can begin on the rising edge of the S-Stage clock.

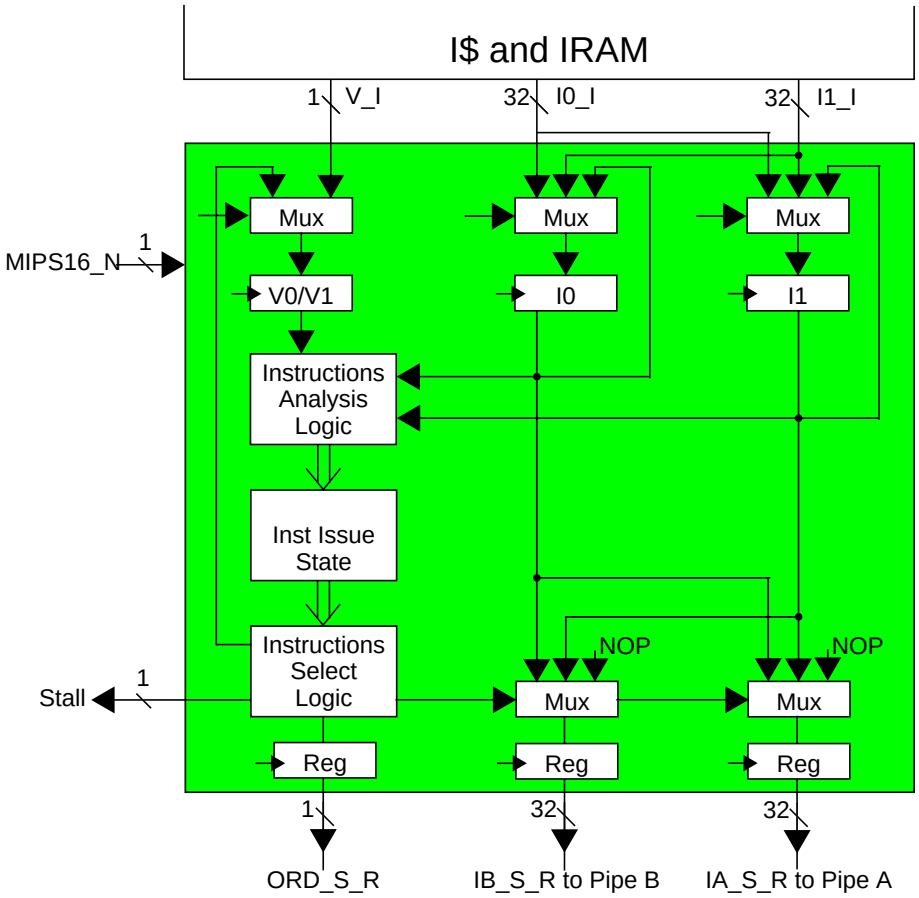


Figure 3: Superscalar Instruction Issue

2.4. RALU Data Path

2.4.1. Overview

The Superscalar RALU Datapath is illustrated in the Figure. Operations are divided between Pipe A and Pipe B in such a way that the RALU is the only major section of the processor which requires both Pipe A and B instructions. Coprocessor 0, as well as the optional customer-defined Coprocessors 1-3, only require the Pipe A instruction.

To “first approximation” the superscalar RALU is a “doubling” of the LX4180 RALU: it includes an 8-port (4r/4w) general register file with 4-ports (2r/2w) assigned to Pipe A, and 4-ports (2r/2w) assigned to Pipe B. In each Pipe, one write port is dedicated to register file updates from the Data Bus (Loads, MFCz, CFCz - moves from Coprocessor). The remaining three ports (2r/1w) are available for the other operations assigned to that Pipe. As a result, loads, including “twinword” loads of register pairs can dual-issue with any MAC or ALU instruction without register port access restriction.

Each Pipe has an ALU and a nearly-independent control section. Differences occur in the assignment of operations to Pipe A and Pipe B, and in the pipeline features to support superscalar. The pipeline differences in the RALU to support superscalar issue are:

- Data must be forwarded from Pipe A (Pipe B) to Pipe B (Pipe A) when the input to a Pipe B (Pipe A) execution unit requires a result computed earlier in Pipe A (Pipe B). The

forwarding paths are illustrated in Figure 2.

- If both Pipe A and Pipe B operations write the same register, the RALU control examines the instruction order and suppresses the write for the earlier instruction based on program order.

In addition, the RALU interfaces with the dual-MAC as a custom engine; this interface can supply two, 32-bit operands per cycle, and return a single 32-bit operand per cycle. In the case of the dual-MAC, each of the 32-bit operands can be interpreted as two independent 16-bit operands.

2.4.2. Assignment of Instructions to Pipe A, Pipe B.

Table 2 lists the detailed assignment of instructions to Pipe A and Pipe B. Pipe B is called the "MAC Pipe" because it uniquely supports multiply-accumulate, as well as multiply and divide operations. The Dual MAC unit, which is attached to Pipe B as Custom Engine 0 (CE0), includes the accumulator registers (including HI and LO) and therefore also supports the *move to* and *move from* operations which transfer data between these registers and the general register file.

Pipe A is called the "Load/Store Pipe" because it uniquely supports the Load and Store operations. DSP extensions to memory addressing are therefore also unique to Pipe A. These extensions include pointer post-modification and circular buffer addressing. The Figure illustrates the circular buffer start registers (cbs0-cbs2) and circular buffer end registers (cbe0-cbe2) located in ALU A.

The Coprocessor operations, and all "sequencing control instructions" (branches, jumps) are unique to Pipe A. As a result, Pipe B instructions are not routed to Coprocessors.

The opcodes reserved for a customer defined Custom Engine 1 (CE1) are routed to Pipe B, since CE1 is attached to Pipe B.

All ALU operations are available in both Pipe A and Pipe B. As a result performance is improved, particularly in computation-intensive programs, and, the design is simplified because major sub-blocks in ALU A and ALU B are replicated.

The Custom Engine Interface (CEI) is available for customer proprietary operations in Pipe B. This allows the customer extensions to maintain high throughput since they can dual-issue with Load and Store instructions which issue to Pipe A.

Table 2: Assignment of Instructions of Pipe A, Pipe B

	Pipe A	Pipe B
	The Load/Store Pipe	The MAC Pipe
MIPS 32-bit General Instructions	MIPS 32-bit General Instructions except: CE1 Custom Engine Opcodes, MULT(U), DIV(U), MFHI, MFLO, MTHI, MTLO,MAD(U),MSUB(U)	MULT(U), DIV(U), MFHI, MFLO, MTHI, MTLO,MAD(U),MSUB(U) CE1 Custom Engine Opcodes, MIPS 32-bit ALU Instructions Note: No Load or Store Instructions
MIPS 32-bit Control Instructions	J, JAL, JR, JALR, JALX SYSCALL, BREAK, All Branch Instructions, All MFCz, MTCz, SWCz, LWCz	

	Pipe A	Pipe B
	The Load/Store Pipe	The MAC Pipe
MIPS16 Instructions (No Doubleword Instructions)	All MIPS16 Instructions except: MULT(U), DIV(U), MFHI, MFLO	MULT(U), DIV(U), MFHI, MFLO
EJTAG Instructions	DERET, SDBBP (including MIPS16 SDBBP)	
Lexra Control Instructions	MTRU, MFRU, MTRK, MFRK, MTLXC0,MFLXC0	
Lexra Vector Addressing	LT, ST, LTP, LWP, LHP(U), LBP(U), STP, SWP, SHP, SBP	
Lexra MAC Instructions		MTA2, MFA, MFA2, MULTA, MULTA2, MULNA2, CMULTA, MADDA, MSUBA, ADDMA, SUBMA, DIVA, RNDA2
Lexra Extensions to MIPS ALU Instructions	SLLV2, SRLV2, SRAV2, ADDR, ADDR2, SUBR, SUBR2, SLTR2	SLLV2, SRLV2, SRAV2, ADDR, ADDR2, SUBR, SUBR2, SLTR2
New Lexra ALU Operations	MIN, MIN2, MAX, MAX2, ABSR, ABS2, CLS, MUX2, BITREV, CMVEQZ, CMVNEZ	MIN, MIN2, MAX, MAX2, ABSR, ABS2, CLS, MUX2, BITREV, CMVEQZ, CMVNEZ

2.5. 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 LX5280 throughput by several percent. However, the net effect of the LX5280’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 3: 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) 0x0000c601 for LX5280
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 LX5280 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 7.2, 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.)

2.6. Dual Multiliply-Accumulate (MAC)

2.6.1. Dual MAC Operations

The Dual MAC data path is illustrated in Figure 4 on page 16. The major subsystems are:

- Two 16-bit Multiply-Accumulate data paths each with:
  - 16-bit x 16-bit Multiplier
  - 32-bit Product Register
  - Four 40-bit Accumulator Registers with optional saturate
  - Output Scalers
- 40-bit Add/Subtract/Dual Round Unit with optional saturate
- One or two 16-bit x 16-bit multiply or multiply-accumulate operations can be initiated every cycle, with a three cycle latency.
- 32-bit x 32-bit multiply executes on a single Multiply-Accumulate data path with five cycle latency. By using both data paths, a 32-bit x 32-bit multiply-accumulate can be initiated every other cycle.
- 32-bit x 32-bit multiply-accumulate executes using a combination of a single Multiply-Accumulate data path, followed by the Add-Round. The total latency is six cycles. By using both data paths, a 32-bit x 32-bit multiply-accumulate can be initiated every other cycle.
- Complex Multiply (16-bit Real, 16-bit Imaginary per-product) uses both Multiply-Accumulate data paths with three cycle latency. A new Complex Multiply can be initiated every two cycles.

- One Divide unit.

2.6.2. MAC MODE (MMD) register

Several new DSP features are controlled using the MMD ("MAC Mode") register. MMD is a new Radiax User register (24) which is accessed using Radiax User Move instructions MTRU and MFRU. If MMD is updated between a MAC instruction and the MFA instruction that retrieves the result of that instruction, the resulting operation is undefined.

The fields in MMD are as follows. Note that MMD is reset to all zeroes.

MF selects arithmetic mode for multiplies in the Dual MAC:

0: use integer arithmetic mode
1: use fractional arithmetic mode

MS selects saturation boundary in the Dual MAC accumulators:

0: saturate at 40 bits
1: saturate at 32 bits

MT selects truncation of 32x32 multiplies in the Dual MAC:

0: perform full 32x32 multiply (sum all four partial products)
1: omit partial product rS[15:00] x rT[15:00] when performing 32x32 multiply.

RND selects the rounding mode used in the *RNDA2* instruction.

00: Convergent Rounding. (Sometimes called "round-to-nearest-even") Round to nearest number; when the number to be rounded is midway between two numbers representable in the smaller format, round to the even number. The rounded result will always have 0 in the lsb. Assuming that the lsb left of the roundoff point is random, convergent rounding is unbiased.

01: Round-to-Nearest
Round to nearest number; when the number to be rounded is midway between two numbers representable in the smaller format, round to the more positive number. (This rounding mode is common because it is easily implemented by always adding 0...0^10...0 to the number to be rounded. Digits to the right of "^" are dropped after rounding.)

1x: reserved

MMD (Radiax User Register 24)

- A new Radiax User register (24)
- Accessed with MTRU, MFRU operations
- Reset to 0

Table 4: MMD Fields (Radiax User Register 24)

Field	Width (Bits)	Description
RND	2	Rounding mode 00 = convergent 01 = round-to-nearest 10 = reserved 11 = reserved
MF	1	MAC fractional mode
MS	1	MAC 32-bit saturate mode
MT	1	MAC 32x32 truncate mode

2.6.3. Architecture

The Multiply-Accumulate data paths can operate on 16-bit input data, either individually or in parallel. The same Assembler mnemonic is used for individual or parallel operation. The output register specified determines whether MAC0 or MAC1 or both, operate. For example,

```
MADDA2  m0l, r2, r3     // MAC0:   m0l ← m0l  + r2[15:00] * r3[15:00]
                        // MAC1:   IDLE

MADDA2  m0h, r2, r3     // MAC0:   IDLE
                        // MAC1:   m0h ← m0h + r2[31:16] * r3[31:16]

MADDA2  m0, r2, r3      // MAC0:   m0l ← m0l  + r2[15:00] * r3[15:00]
                        // MAC1:   m0h ← m0h + r2[31:16] * r3[31:16]
```

Each Multiplier can initiate a new 16-bit x 16-bit product every cycle (*single cycle throughput*). Each 16-bit x 16-bit multiply-accumulate completes in three cycles. (Figure 4 illustrates the intermediate pipeline registers TEMP0, TEMP1, Product 0, Product 1 to help the reader remember that the Multipliers require two cycles but have single cycle throughput. TEMP0, TEMP1, Product 0, Product 1 are not accessible by the programmer.). Thus, there are *two* delay slots for Multiplication or Multiply-Accumulate. For example,

```
Cycle 1:    MADDA2      m1h, r2, r3
Cycle 2:    delay slot 1                // new m1h is not available
Cycle 3:    delay slot 2                // new m1h is not available
Cycle 4:    MFA         r3, m1h         // new m1h is available
```

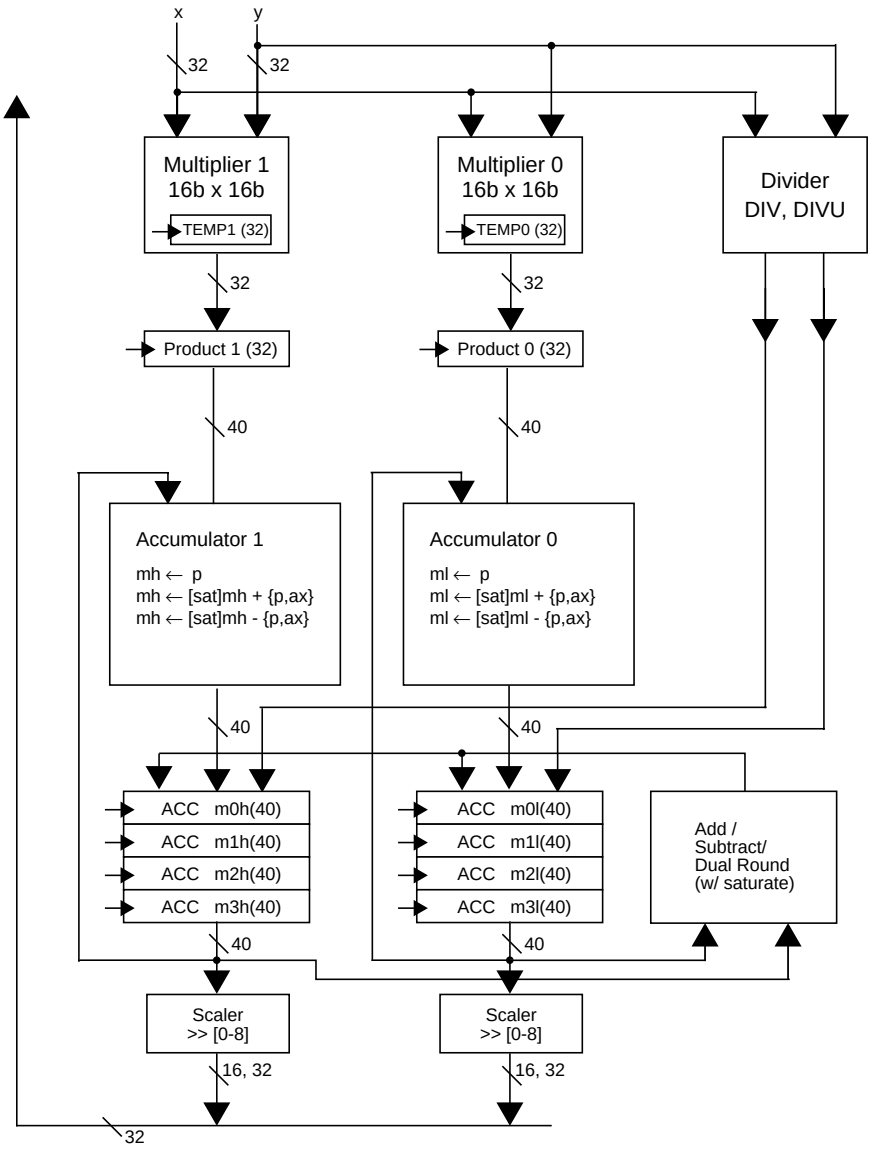


Figure 4: Dual MAC Data Path

The accumulator m1h can be referenced by MFA in Inst2(Inst3), however two (one) stall cycles will be incurred. It is expected that the number of stall cycles in DSP algorithms will be minimal because, typically many products are accumulated before the accumulator must be stored. In a 64-tap FIR, for example, 64 terms are accumulated before the filter sample is updated in memory. Also, the four accumulator pairs allow loops to be "unrolled" so that up to three additional independent MAC operations can be initiated before the result of the first is available.

Compared to a typical RISC multiply-accumulate unit the LX5280 MAC includes a number of features critical to high-fidelity DSP arithmetic. These features are optionally selected by opcodes and/or mode bits in the MMD register, and are compatible with conventional integer arithmetic, also supported by the LX5280:

- Accumulator guard bits,
- Fractional arithmetic,
- Saturation,
- Rounding,
- Output Scaling.

Accumulation is performed at 40-bit precision, using eight *guard bits* for overflow protection. The alternative is to require the programmer to right-shift (scale) products prior to accumulation, which complicates programming and causes loss of precision. Prior to accumulation, the product is sign-extended to 40-bits. With guard bits, typically the only loss of precision will occur at the end of a lengthy calculation when the 40-bit result must be stored to the general register file or to memory in 32-bit or 16-bit format.

Fractional arithmetic is implemented by the program's interpretation of the 16-, 32- or 40-bit quantities and is controlled by a bit in the MMD register. When fractional mode is selected, the Dual MAC shifts the results of any Radiax multiply operation left by one bit to maintain the alignment of the implied radix point. Furthermore, since -1 can be represented in fractional format but +1 cannot be represented in fractional mode, the Dual MAC detects when both operands of a multiply are equal to -1. If so, it generates the approximate product consisting of 0 for the sign bit (representing a positive result) and all ones for the remaining bits. This is true for both 16x16 bit and 32x32 bit Radiax multiplications. The least significant bit of a product is always zero in fractional mode (due to the left shift).

The Accumulation Units can add the product to, or subtract it from, one of the four accumulator registers. This operation can be performed with optional *saturation*; that is, if a result overflows (underflows), the accumulator is updated with the largest (smallest) positive (negative) number rather than the "wraparound" result with incorrect sign. The LX5280 instructions include a Multiply-add and Multiply-sub, each with and without saturation. There are also instructions for adding or subtracting any pair of 40-bit accumulator registers together, with and without saturation. A bit in the MMD register determines whether the saturation is performed on the full 40 bits or whether saturation is performed at 32 bits. The latter capability is useful for emulating the results of other architectures that do not have guard bits. In 32-bit saturation mode, a full 40- bit compare is used to determine if the result is greater (less) than the maximum (minimum) value which can be stored in a 32-bit quantity; this provides the most robust solution.

In the case that the instruction requires multiplication, but no accumulation, the product is passed through the accumulation unit unchanged. (Thus, both 16-bit multiplication and multiply-accumulate require three MAC cycles.)

A Round instruction can also be executed on one (or a pair) of the accumulator registers to reduce precision prior to storage. The rounding mode is selectable in the MMD register.

The output Scaler is used to right shift, (scale), the accumulator register when it is transferred to the general register file.

The Dual MAC is also used to execute the 32-bit MULT(U) and DIV(U) instructions specified in the MIPS ISA. In the case of MULT(U), one of the 16-bit Multiply-Accumulate data paths works iteratively to produce the 64-bit product in five cycles. (The least significant 32 bits are available one cycle earlier than the most significant 32 bits.)

Note: The MMD mode bits have no effect on the operation of the standard MIPS ISA instructions. By contrast, the LX5280 MULTA instruction is subject to the MMD mode bits for fractional arithmetic and truncated 32x32 multiplication.

32-bit x 32-bit Multiply-Accumulate instructions (MADDA, MSUBA) are implemented using one of the 16-

bit Multiply-Accumulate data paths, and the Add-Round unit. It provides a 64-bit Multiply result which is sign-extended and accumulated at 72-bits. The result is available in six cycles. (The least significant 32 bits are available one cycle earlier than the most significant 40 bits.) The MAD(U) and MSUB(U) instructions of the MIPS32 ISA are also supported.

For the LX5280 MULTA, an accumulator pair M0h[39:0]/M0l[31:0], M1h[39:0]/M1l[31:0] etc. is the target. M0h[39:0] is aliased to HI; M0l[31:0] is aliased to LO. The most significant 8-bits of the 40-bit HI accumulator are used as the guard bits, while the LO accumulator is simply zero-extended to 40 bits. Unlike the (dual) 16-bit operations, single-cycle throughput is not available for 32-bit data. However since there are two available data paths, two 32-bit x 32-bit multiply operations can be initiated every four cycles. The Dual MAC hardware automatically allocates the second operation to the available data path. If a third 32-bit multiplication is programmed too soon, stall cycles are inserted until one of the data paths is free.

The Dual MAC also supports a complex multiply instruction, CMULTA. For this instruction, each of the 32-bit general register operands is considered to represent a 16-bit real part (in bits 31:16) and a 16-bit imaginary part (in bits 15:00). One of the multiply-accumulate engines calculates the real part (33 bits) of the complex product (namely XrYr - XiYi) and stores it in the “h” half of the target accumulator pair. The other MAC engine calculates the imaginary part (32 bits) of the complex product (namely XrYi + XiYr) and stores it in the “l” half of the target accumulator pair. This instruction can be initiated every two cycles (2-cycle throughput) and takes four cycles to complete. As in the other Dual MAC operations, programming CMULTA instructions too close together causes stall cycles but the correct results are always obtained.

The Dual MAC includes a separate Divide Unit for executing the 32-bit DIV(U) operations specified by the MIPS ISA. The Divide requires 19 cycles to complete. The quotient is loaded into M0l[31:0], M1l[31:0], M2l[31:0] or M3l[31:0] and the remainder is loaded into the lower 32-bits of the other accumulator in the target pair. There is no special support for fractional arithmetic for the divide operations.

2.7. Data Addressing

2.7.1. Twinword Data Movement

Since the Dual MAC is capable of consuming four 16-bit operands every cycle (in Pipe B) by performing two 16x16 multiply-accumulates, it is desirable to be able to fetch four 16-bit operands from memory every cycle (in Pipe A). Therefore, the LX5280 extends the MIPS load and store instructions to include twinword accesses and implements a 64-bit data path from memory. A twinword memory operation accesses an (even-odd) pair of 32-bit general registers with a single instruction and executes in a single pipeline cycle. The nomenclature “twinword” is used to distinguish these operations from “doubleword” operations which (in other extensions to the MIPS ISA) access a single 64-bit general register.

Like the standard byte, halfword, and word load/store instructions, the twinword load/store instructions use a register and an immediate field to specify the memory address. However, in order to obtain the maximum range from the LEXOP instruction format, the available signed 11-bit immediate field (called the displacement) is considered a twinword quantity, so is left-shifted by 3 bits before being added to the base register. This is equivalent to a 14-bit byte offset, in comparison to the full 16-bit immediate byte offset used in the byte, halfword and word instructions. Also, the target register pair for the twinword load/store must be an even-odd pair, so that only 4 bits are used to specify it.

2.7.2. Vector Addressing

DSP algorithms usually operate on vectors or matrices of data; for example Discrete Cosine Transforms operate on 8x8 pixel blocks. As a result data memory pointers are incremented from one operand to the next. The extra instruction cycle required to increment RISC memory pointers is eliminated in DSPs with auto-increment. This capability is provided in the LX5280. Memory pointers are used unmodified to create the

address, then updated in the general register file before the next use:

```
address   ←   pointer
pointer   ←   pointer + stride
```

In the LX5280 the 8-bit immediate field containing the stride is sign-extended to 32-bits before being added to the pointer for the latter's update. The nomenclature "pointer" is used to distinguish the update performed *after* memory addressing from the "offset", in which the "base" register (in the MIPS ISA) which is augmented by the offset *before* addressing memory in the standard instructions. The nomenclature "stride", which is dependent on the granularity of the access, is used to distinguish it from the invariant byte offset used in the standard load and store instructions. For twinword/word/halfword addressing the 8-bit field is first left-shifted by three/two/one places and zero-filled, before sign extension to 32-bits. This use of left shifts for the twinword, word, and halfword word and halfword strides is similar to MIPS16 and is used to extend the effective address range. Thus, increments of between -128 and +127 twinwords[1], words, halfwords or bytes are available for each data type.

In the case of Loads (but not Stores) pointer update requires a second general register file write port. The LX5280 includes an 8p(4r/4w) register file with two of the four write ports dedicated to register Loads. As a result, twinword loads can execute in parallel with any Pipe B operation.

For some DSP algorithms - notably Filters - DSP data is organized into "circular buffers". In this case, at the end of the buffer the next reference is to the beginning of the buffer. Implementing this structure in RISC requires:

```
Inst 1:     LW      reg, AddressReg
Inst 2:     BNEL    AddressReg, BufferEnd, Continue
Inst 3:     ADDIU   AddressReg, AddressREG + 4
Inst 4:     MOVE    AddressReg, BufferStart
Continue:
```

Note that the above example is written so that a branch prediction failure will only be incurred at the end of the buffer. Nevertheless, the combination of post-modified pointers together with hardware support for circular buffers in the LX5280 allows this typical DSP addressing operation to be reduced from four cycles to one.

1. Twinwords are supported only on the LX5280, and not the LX5180

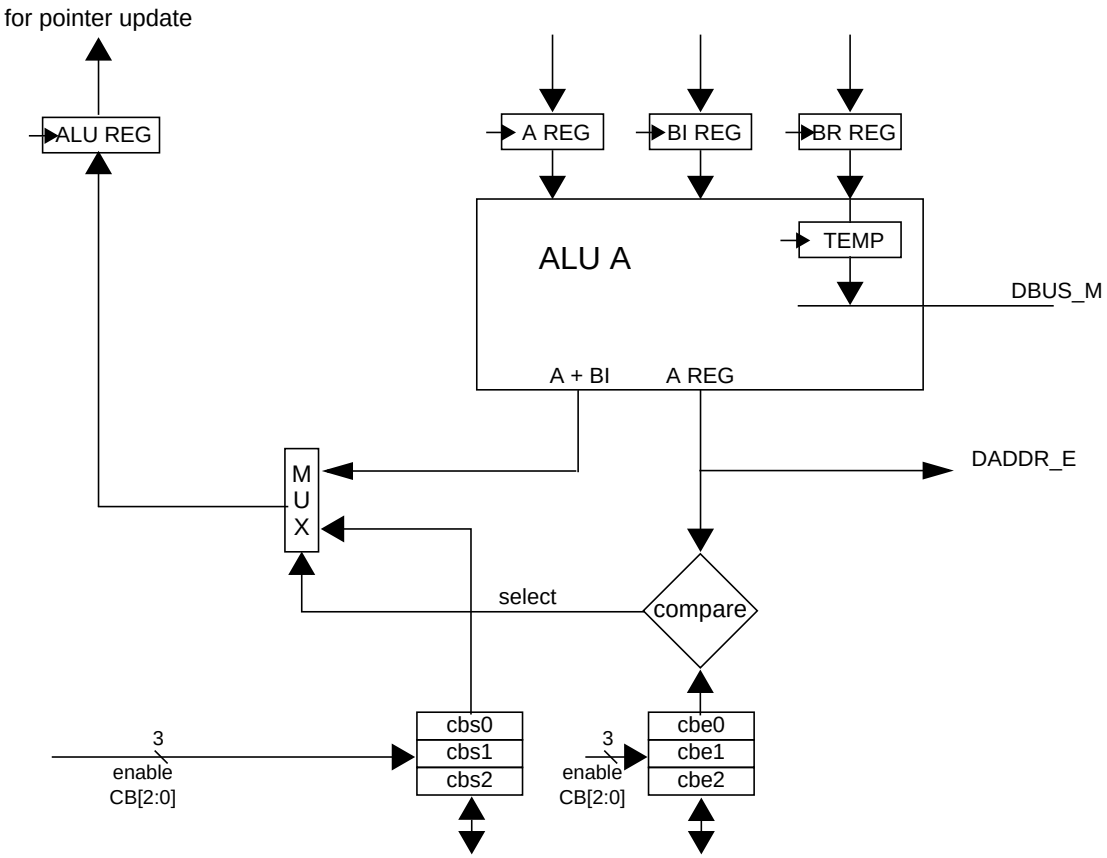


Figure 5: Post-modified Pointers with Circular Buffer Support

2.7.3. Circular Buffers

The LX5280 supports three circular buffers. To initialize the circular buffers, the MTRU instructions are used to set the twinword start addresses CBS0-CBS2[31:3] and twinword end addresses CBE0-CBE2[31:3]. Circular buffers are only used when memory pointers are post-modified, and consist of an integral number of twinwords.

When a circular buffer pointer is used in a post-modified address calculation, the pointer is compared to the associated CBE address; if they match (and the stride is non-negative), the CBS address (rather than the post-modified address) is restored to the register file. Similarly, to allow for traversing the circular buffer in the reverse direction, the pointer is compared to the CBS address; if they match (and the stride is negative) the CBE address (rather than the post-modified address) is restored to the register file.

It is worth noting that circular buffers can also be accessed with byte, halfword, or word Load/Store with Pointer Increment instructions. In those cases, the several least significant bits of the pointer register are examined to determine if the start or end of the buffer has been reached, taking into account the granularity of the access, before replacing the pointer with the CBS or CBE as appropriate.

Any general register memory pointer can be used with *circular buffers* using the ".Cn" option. To use general register rP as a circular buffer pointer. For example, the instruction

```
LWP.C2          r3, (r4)stride
```

associates the r4 memory pointer with circular buffer C2 which is defined by the start address CBS2 and end address CBE2.

2.8. Radiax ALU Operations

The LX5280 introduces extensions to the MIPS instructions to support dual 16-bit operations. The LX5280 also introduces a number of new ALU instructions which improve performance on DSP algorithms. These instructions will also be described in this section.

2.8.1. Extensions to MIPS ALU Operations

To support high-performance dual 16-bit operations in the RISC-DSP it is necessary to support not only Dual MAC instructions but also dual 16-bit versions of other arithmetic operations that the programmer may require. To maintain a simple, orthogonal instruction set, the following criteria were used to determine the MIPS ALU extensions:

- Dual 16-bit versions of *all* MIPS ALU operations without immediate data,
- Optional saturation for every ALU instruction (without immediate data) that can produce signed overflow or underflow.

It is expected that the above organizing principles will simplify the LX5280 ISA for both programmers and tool developers. Obviously, dual 16-bit versions of logical operations such as AND are not required. However, dual 16-bit versions have been provided for all 3-register operand shifts and add/subtracts included in the MIPS R-Format. The character “2” in the assembler mnemonic indicates an operation on dual 16-bit data.

DSP algorithms are often somewhat tolerant of data errors. For example, a bad audio sample may cause a brief distortion, but no lasting effect as new audio samples arrive and the bad sample is cleared out of the buffer. Accordingly, the saturated result of signed arithmetic is a closer, more desirable, approximation than the wraparound result. Therefore, all LX5280 arithmetic operations which may, potentially, produce arithmetic overflow or underflow, and do not have immediate operands, support optional saturation. For example, not only the dual 16-bit add (ADDR2) but also the 32-bit add (ADDR) have optional saturate in the LX5280. Saturation options are not provided for MIPS I-Format 32-bit instructions; for example ADDIU. However, in this case the programmer selects the immediate operand and, as a result, saturation is less likely, or at least more predictable.

Neither the dual 16-bit instructions nor the new 32-bit saturating adds and subtracts cause exceptions.

2.8.2. New ALU Instructions

The LX5280 adds several new ALU instructions which have proven useful in DSP performance analysis. Consistent with the approach described above, each new instruction has both a 32-bit and a dual 16-bit version. If signed overflow/underflow is possible, a saturation option is provided.

2.8.3. Conditional Move Operations

The LX5280 includes new instructions (MOVZ and MOVN) to support conditional operations. These instructions are described in this section.

A number of DSPs and RISC processors have deployed extensive “conditional execution.” In these processors the branch prediction penalty is three cycles or more. Conditional execution can mitigate the effect of the branch prediction penalty by allowing the branch to be avoided in some cases. However, conditional execution is a costly alternative: it uses instruction opcode bits and consequently limits the size of immediate

data and/or limits the number of general purpose registers visible to the program. The LX5280 branch prediction penalty is only two cycles; therefore the need for conditional execution is minimized and only a restricted set of "conditional move" instructions is needed. It is notable, however, that the effect of any conditional execution can be "emulated" in the LX5280 with a sequence of two instructions by using the conditional move. For example:

Processor with conditional execution:

```
Inst 1:   ALU operation sets condition flags
Inst 2:   COND: ALU operation
```

LX5280:

```
Inst 1:   ALU operation updates register rB (condition setting operation)
Inst 2:   ALU operation with result directed to temp register rA
Inst 3:   MOV<COND>  rD, rA, rB
```

If rB satisfies the COND, rD is updated with rA; i.e. the 2nd ALU operation is executed to "completion". Note that this sequence is interruptible.

Another use of the conditional move instructions is to code "if-then-else" constructs as follows:

```
if (rB COND)
        rD = rA
else
        rD = rC
```

can be coded if the previous example is prefaced with:

```
MOVE rD, rC             // move rC to rD
```

One reason Lexra has provided conditional move is to facilitate initial porting of Assembler code from processors with conditional execution to the LX5280.

2.9. Zero Overhead Loop Facility

Because DSP algorithms spend much of their time in short real-time critical code loops, DSPs often include hardware support for "zero-overhead looping." The goal of zero-overhead looping is that branching from the end-to-beginning of the loop can be accomplished without explicit program overhead if the loop is to be executed a fixed number of times, known at compile time.

The LX5280 supplies such a facility but allows the loop count to be determined at run time as well. The facility consists of three new Radiax registers, which are accessible by a program running in User mode using the Radiax instructions MFRU and MTRU. The operating system should consider these registers as part of the context of the executing process and must save and restore them in the case of an interrupt.

LPE0[31:2] —virtual address of the ending instruction of the loop

LPS0[28:2] —low order bits of the virtual address of the "starting" instruction of the loop.

LPC0[15:0] —the loop count.

Although the facility is intended for use in loops, the algorithm executed by the hardware can be described more simply. In particular it should be noted that there is no "knowledge" of being "inside" the loop. All that matters is the contents of the three registers when an attempt is made to execute the instruction at the address

specified by LPE0:

```
if (M32-mode, AND current-instruct-addr[31:2] = LPE0, AND LPC0 ≠ 0) then
    execute current instruction (at LPE0[31:2] || 00),
    decrement LPC0[15:0] by one,
    execute instruction at LPE0[31:29] || LPS0[28:2] || 00
    continue (LPS0 could be a jump/branch)
else
    execute current instruction,
    continue (current instruction could be a jump/branch)
```

The following restrictions apply to the usage of the Zero Overhead Loop Facility:

- It is only active in 32-bit ISA mode. It is disabled in MIPS16 mode.
- LPS may not be exactly equal to LPE if LPC is non-zero. Therefore, the loop must contain at least two instructions. Otherwise, operation is undefined.
- LPE may not be in the delay slot of a branch, nor may it be a branch or jump instruction itself if LPC is non-zero. Otherwise, operation is undefined.
- For correct operation, the order of loading the registers must be: first LPS, then LPE, then LPC with a non-zero value.
- For correct operation, there must be at least two (2) instructions between the instruction which loads LPC with a non-zero value, and the instruction at the LPE address. To guarantee that no stall cycles are incurred, there must be at least three (3) cycles between the instruction which loads LPC with a non-zero value, and the instruction at the LPE address.[1]
- If the instruction at LPE is a load type instruction, then the immediately executed instruction at LPS is considered to be in the load delay slot and cannot rely on seeing the result of the load.

The following items are *not* restrictions that apply to the usage of the Zero Overhead Loop Facility but are features to be aware of:

- The loop count LPC may be reloaded multiple times after LPS and LPE are loaded. Typically this would be done in an outer loop.
- The instruction at LPE may be the target of a jump or branch, including a change in mode from 16-bit to 32-bit ISA.
- Any of the instructions before or at LPE may be subject to exceptions or interrupts and processing will conform to the normal exception handling rules. Note that the BD bit will

1. The following discussion is only relevant if LPC will be updated in an instruction that is “close” to LPE. That case can have a performance impact although correct operation will still be obtained. The programming guideline is: keep the LPC update in an outer loop as far as possible from the (end of) the inner loop:

   The updates of LPS, LPE, and LPC use the MTRU instruction. Therefore the new LPS, LPE, and LPC values are only known after the E-stage of the pipeline. But in order to perform the pseudo-branch they must be used in the I-stage of the pipeline. Because of the restriction on the order of setting these registers, the hardware introduces a minimum number of stalls after setting LPS and LPE to test for an LPE match against the current instruction address. However, if the LPC update is still in the pipeline when the LPE match is detected, the hardware stalls to check and update the new value of LPC. To avoid these stalls, LPC should not be updated within 3 cycles (which could be as many as 6 instruction issue slots) of an expected LPE-matching instruction. As noted above, for correct operation there must be at least 2 instructions between the LPC update and any expected LPE-matching instruction.

always be off since LPE must not be in the delay slot of a branch. The return from the exception handler to LPE will also be handled normally, since it is just a special case of LPE being the target of a jump.

- If the instruction at LPE causes a Reserved Instruction Trap, it is necessary for the Exception Handler to decrement LPC prior to return, after emulating the instruction at LPE and before returning to the instruction at LPS. Similar restrictions apply if the instruction at LPE is not to be re-executed for any other reason, such as BREAK or SYSCALL execution.

2.10. Low-Overhead Prioritized Interrupts

The LX5280 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 processor concatenates the program defined BASE address for the exception vectors with the interrupt number for form the interrupt vector, as shown in the table below. Two instructions can be executed in each vector; typically these will consist of a jump instruction and its delay slot, with the target of the jump being either a shared interrupt handler or one that is unique to that particular interrupt.

Table 5: Prioritized Interrupt Exception Vectors

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

When a vectored interrupt causes an exception, all of the standard actions for an exception occur. These include updating the EPC register and certain 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. LX5280 RISC Programming Model

This section describes the LX5280 Programming Model. Section 3.1, Summary of MIPS-I Instructions, contains a list summarizing all MIPS-I operations supported by the LX5280. 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 LX5280 LMI. The SMMU provides sufficient memory management capabilities for most embedded applications while ensuring execution of third-party MIPS software development tools.

The LX5280 supports the MIPS R3000 Exception Processing model, as described in Section 3.4, Exception Processing.

The LX5280 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 LX5280. The CI is described in Section 3.5, The Coprocessor Interface (CI).

3.1. Summary of MIPS-I Instructions

The LX5280 executes MIPS-I instructions as detailed in the tables below. To summarize, the LX5280 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 following conventions are employed in the instruction descriptions.

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

3.1.1. ALU Instructions

Table 6: ALU Instructions

Instruction		Description
ADD ADDU ADDI ADDIU	rD, rA, rB rD, rA, rB rD, rA, immediate rD, rA, immediate	rD <- rA + «rB, immediate» Add reg rA to either reg rB or a 16-bit immediate sign-extended to 32 bits. Result is stored in reg rD. ADD and ADDI can generate overflow trap; ADDU and ADDIU do not.
SUB SUBU	rD, rA, rB rD, rA, rB	rD <- rA - rB Subtract reg rB from reg rA. Result is stored in register rD. SUB can generate overflow trap. SUBU does not.
AND ANDI	rD, rA, rB rD, rA, immediate	rD <- rA & «rB, immediate» Logical *and* of reg rA with either reg rB or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
OR ORI	rD, rA, rB rD, rA, immediate	rD <- rA | «rB, immediate» Logical *or* of reg rA with either reg rB or a 16-bit immediate zero-extended to 32 bits. Result is stored in reg rD.
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'b0} 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 <- (rA < «rB, immediate») ? 1 : 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 rD, offset(rA) LBU rD, offset(rA) LH rD, offset(rA) LHU rD, offset(rA) LW 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.
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	rD <- (rT== 0) ? rS : rD If the contents of general register rT are equal to 0, the general register rD is updated with rS; otherwise rD is unchanged.
MOVN rD, rS, rT	rD <- (rT != 0) ? rS : rD If the contents of general register rT are not equal to 0, the general register rD is updated with rS; otherwise rD is unchanged.

3.1.4. Branch and Jump Instructions

Table 9: Branch and Jump Instructions

Instruction		Description
BEQ BNE	rA, rB, destination rA, rB, destination	if COND pc <- pc + 4 + { 14 { (destination[15] }, destination, 2'b00 } 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.
BLEZ BGTZ	rA, destination rA, destination	if COND pc <- pc + 4 + { 14 {destination[15] }, destination, 2'b00 } 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, 2'b00 } 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, 2'b00 } 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 when SYSCALL is executed.
BREAK	The Bp Trap occurs when BREAK is executed.
RFE	Causes the KU/IE stack to be popped. Used when returning from the exception handler. See “Exception Processing” below.
SLEEP	Initiates low-power standby mode. This is a Lexra specific operation (LEXOP). See Section 3.6, Power Savings Mode.

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	if COND pc <- pc + 4 + { 14’ { destination[15] } , destination, 2’b00 } else pc <- pc + 8 where COND = (CpCondz = True) for BCzT, (CpCondz = False) for BCzF. For BCzT, BCzF the instruction after the branch (*delay slot*) is always executed.

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

3.2.1. CEI Operations

Customers may add proprietary or application-specific opcodes to their LX5280 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	I/O	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.

Signal	I/O	Description
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 LX5280 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

3.4. Exception Processing

The LX5280 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 LX5280 Core. Each is synchronized by the Core to the LX5280 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: 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 LX5280, user mode virtual addresses must have msb = 0. In kernel mode, the full address space is addressable. IE = 1(0) indicates that interrupts are enabled (disabled).
The KUo, IEo, KUp, IEp, KUc and IEc fields form a three-level stack hardware stack KU/IE signals. The *current* values are KUc/IEc, the *previous* values are KUp/IEp, and the *old* values (those before previous) are KUo/IEo. (See Section 3.4.2.)

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 LX5280.

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 } (before push)

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

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

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

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

```
MFC0    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 LX5280. 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 LX5280. 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 9, LX5280 Coprocessor Interface.

3.6. Power Savings Mode

The operating system kernel can initiate a power savings standby mode using the Lexra specific SLEEP

instruction. This holds the LX5280's internal clocks in the high state until an external hardware interrupt is received.

Before executing the SLEEP instruction, the kernel must ensure that the interrupt condition that will ultimately terminate standby mode has been enabled via the IM field of the coprocessor 0 Status register. When the SLEEP instruction enters the W stage, the standby logic stalls the processor and waits for the LBC to complete any outstanding processor initiated system bus operations. After these are completed, the standby logic holds the system and bus clocks high. These are held high until an enabled interrupt is received.

When standby mode is terminated by an interrupt, the standby logic allows the clocks to toggle. The processor honors the interrupt by branching to the exception handler as is normally done for interrupt servicing. Because several instructions are held in the pipeline while the clocks are frozen prior to the interrupt, the exception PC will not point to the SLEEP instruction, but rather some later instruction. Typically, a kernel would enter an idle loop just after executing the SLEEP instruction, so the interrupt will be serviced from the kernel's normal idle interrupt service level.

The LX5280 takes a minimum of 6 cycles after the SLEEP instruction enters the W stage to safely synchronize the initiation of standby mode, i.e. hold the clocks in the high state. Two cycles are required terminate standby mode. The processor is stalled during these periods.

The standby logic receives the free running system and bus clocks, and generates gated clocks for distribution to the LX5280. The standby logic must use flip-flops tied to free running clocks, which results in about a dozen loads on the free running clocks.

Two pins, SL_SLEEPING_R and SL_SLEEPING_BR, are available from the standby logic and are asserted high when the processor is in standby mode. The _R pin is for use in the system clock domain, and the _BR pin is for use in the bus clock domain.

4. MIPS16

MIPS16 is an extension to the MIPS Instruction Set Architecture (ISA) that was developed to improve code density, especially for System-on-Chip (SoC) designs. In these designs, on-chip instruction storage is often a significant, even dominant, portion of the silicon component cost. This is especially true for real-time applications because, in order to meet real-time requirements, instruction cache miss penalties cannot be tolerated and thus a large portion of the instruction storage must be resident on-chip.

MIPS16 provides a set of 16-bit instruction formats to encode the most common operations. The key compromises required to achieve 16-bit encoding include: (i) some MIPS I instructions are not available, (ii) immediate widths are reduced, (iii) only 8 of the 32 general registers may be directly addressed. As a result some operations cannot be executed in MIPS16 or require multiple MIPS16 instructions. Thus realistic programs need to include both MIPS16 and MIPS I instructions, using MIPS16 where possible to save storage, at some cost to performance.[1] Mode switching between MIPS16 and MIPS I is discussed below. To permit occasional access to all 32 general registers without the overhead of mode switching, MIPS16 provides *MOVE* instructions to move data between the MIPS16-visible registers and the full general register set. Also, to permit occasional use of 16-bit immediates without mode switching, MIPS16 provides the *EXTEND* instruction to allow a full width immediate in two MIPS16 instruction cycles. (Programs requiring a large register set or frequent full-width immediates should be compiled in MIPS I.)

MIPS16 is difficult to program effectively at the assembler level. This is because of the limited register set and the restricted size immediates. In fact, according to Sweetman[2], "MIPS16 is not a suitable language for assembly coding". Rather, MIPS16 is viewed as a compiler option which can be effectively applied to achieve significant code size reduction where performance is not critical.

4.1. MIPS16 Instructions

This section describes the MIPS16 instructions, with emphasis on the differences between MIPS16 and the 32-bit MIPS ISA. The first table lists MIPS I Instructions that are *not supported* in MIPS16.

The second table lists MIPS I instructions *which are supported* in MIPS16. In most cases these are specialized versions of the MIPS I instruction. MIPS16 is compatible with MIPS I, II and III, IV or V. The LX5280 implements *all* MIPS16 for 32-bit data operations.[3] The table lists all MIPS16 instructions together with the corresponding MIPS I instruction and the specialization required to produce the MIPS16 instruction (other than smaller register set and smaller immediates).

The third table lists the several new instructions introduced by MIPS16.

It is notable that MULT(U), DIV(U) are supported in MIPS16. MFHI and MFLO are also supported and are necessary to access the result of MULT(U) or DIV(U). However, MTHI and MTLO are not supported. These are used primarily to restore the state after exception handling and are used within the kernel, typically in MIPS I.

1. The MIPS16 performance penalty results from occasionally using two instructions where one MIPS I instruction would suffice. Some of this penalty is recovered in applications where a larger number of instructions per cache line reduces cache miss rate.
2. “See MIPS Run”, Dominic Sweetman, Appendix D, p. 425.
3. MIPS16 includes 16-bit formats for a number of MIPS III 64-bit doubleword operations which are not supported in the MIPS I ISA.They are also not supported in Radiax.

Table 16: MIPS I Instructions Not Supported by MIPS16

MIPS I Not Supported by MIPS16	Assembler Mnemonics
Coprocessor operations	CTCz, CFCz, MTCz, MFCz, LWCz, SWCz, BCzT, BCzF, COPz
Unaligned loads, stores	LWL, LWR, SWL, SWR
Arithmetic operations	ADD, ADDI, SUB
Conditional branches	BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL
Logical operations with immediates	ANDI, ORI, XORI, LUI
Jump	J
Miscellaneous	SYSCALL, RFE, MTHI, MTLO

Table 17: MIPS16 Instructions that Support MIPS I

MIPS16 Instruction	MIPS I Equivalent Instruction[a]

LB(U) ry, offset(rx)
LH(U) ry, offset(rx)
LW ry, offset(rx)
LW rx, offset(sp) (r29 base)
SB ry, offset(rx)
SH ry, offset(rx)
SW ry, offset(rx)
SW rx, offset(sp) (r29 base)
	LW rx, offset(base); base = r29
	SW rx, offset(base); base = r29

ADDIU ry, rx, immediate
ADDIU rx, immediate
ADDIU sp, immediate (1-operand)
ADDIU rx, sp, immediate (2-operand)
ADDU rz, rx, ry
SUBU rz, rx, ry
NEG rx, ry (2-operand)
	ADDIU rt, rs, immediate; rt=rs
	ADDIU rt, rs, immediate; rt=rs=r29
	ADDIU rt, rs, immediate; rs=r29
	SUBU rd, rs, rt; rs=r0

SLT(U) rx, ry (r24 dest. implied) SLTI(U) rx, immediate (2-op., r24 dest)	SLT(U) rd, rs, rt; rd=r24 SLTI(U) rt, rs, immediate; rt=rs
CMPI rx, immediate (r24 dest. implied) CMP rx, ry (r24 dest. implied)	XORI rt, rs, immediate; rt=r24 XOR rd, rs, rt; rd=r24
AND rx, ry (2-operand) OR rx, ry (2-operand) XOR rx, ry (2-operand) NOT rx, ry (2-operand) MOVE ry, r32 (2-operand) MOVE r32, ry (2-operand) LI rx, immediate	AND rd, rs, rt; rd=rs OR rd, rs, rt; rd=rs XOR rd, rs, rt; rd=rs NOR rt, rs, rt; rs=r0 OR rd, rs, rt; rs=r0 OR rd, rs, rt; rs=r0 ORI rd, rs, immediate; rs=r0
SLL rx, ry, immediate SRL rx, ry, immediate SRA rx, ry, immediate SLLV ry, rx (2-operand) SRLV ry, rx (2-operand) SRAV ry, rx (2-operand)	SLLV rd, rt, rs; rd=rs SRLV rd, rt, rs; rd=rs SRAV rd, rt, rs; rd=rs
MULT(U) rx, ry DIV(U) rx, ry MFHI rx MFLO rx	

MIPS16 Instruction		MIPS I Equivalent Instruction[a]	
JAL	target		
JR	rx		
JR	ra	JR	rs; rs=r31
JALR	ra, rx (2-operand; link = r31)	JALR	rs, rd; rs=r31
BEQZ	rx, offset (1-operand)	BEQ	rs, rt, offset; rt=r0
BNEZ	rx, offset (1-operand)	BNE	rs, rt, offset; rt=r0
BTEQ	offset (implied operands)	BEQ	rs, rt, offset; rs=r24, rt=r0
BTNE	offset (implied operands)	BNE	rs, rt, offset; rs=r24, rt=r0
B	offset (implied operands)	BEQ	rs, rt, offset; rs=r0, rt=r0
BREAK			

a. If no 32-bit MIPS instruction is listed, no specialization beyond limited size register set and limited size immediates is required.

As noted earlier, MIPS16 restricts the MIPS I directly addressable register set and immediate field. Another common MIPS16 restriction is that two, rather than three, register operands, are permitted. MIPS16 provides a number of instructions that are not found MIPS I, as shown in Table 18.

Table 18: New MIPS16 Instructions

New MIPS16 Instruction		Comment
LW	rx, offset(pc)	Load word with pc-relative address
ADDIU	rx, pc, immediate	ADDIU with pc operand
EXTEND	immediate	Supplies 11-bit immediate for use in the following MIPS16 instruction
JALX	target	Jump to target, store return in r31 *and toggle the ISA mode between MIPS16 and MIPS I.*

The pc-relative load LW is important to overcoming the drawback of smaller immediates in MIPS16. It allows full 32-bit immediates to be embedded in the program and loaded into registers in a single instruction. The ADDIU with pc operand is useful to support immediates embedded in the program. The pc value referenced in LW or ADDIU depends on the context of the pc-relative instruction as shown in Table 19.

Table 19: PC-Relative Addressing

Context for PC-Relative Instruction	pc Value
Normal case. (Non-extended pc-relative instruction, not in jump delay slot.)	pc of the pc-relative instruction.
pc-relative instruction with extended immediate	pc of the EXTEND instruction
Non-extended pc-relative in the delay slot of jump, JR, JALR, JAL(X) (extended instructions are not permitted in the delay slot of the jump.)	pc of the jump instruction

EXTEND is used to supply an extra 11-bits of immediate. It is used together with the restricted size immediate field of the next instruction to supply a full width immediate. EXTEND cannot occur in the delay slot of a Jump. It is not necessary for the assembly programmer to code EXTEND instructions. It will automatically be assembled by MIPS16 assemblers wherever the immediate is too large to be encoded in a single MIPS16 instruction.

Another new instruction JALX, is available in both MIPS16 and also in MIPS I on machines implementing MIPS16 and is discussed below. [in MIPS I machines not implementing MIPS16, the JALX opcode 000111 causes an RI trap.]

4.2. Mode switching

Mode is switched between MIPS16 and MIPS I in one of two ways:

1. The instruction,

```
JALX    target
```

toggles the mode.

2. The lsb of the general register rx in

```
JR      rx
JALR    rs, rx      (in MIPS16 rs=ra)
```

causes the mode to be set to MIPS16 if rx[0] = 1; to MIPS I if rx[0] = 0. However, the lsb of the instruction memory address from JR/JALR is forced to 0. As a consequence, machines that implement MIPS16 never take AdEL exceptions on the lsb of the instruction address (this is true regardless of whether the machine is operating in MIPS16 or MIPS I mode.).

The mode bit is saved in the lsb of the link register in JAL, JALX, JALR.

4.3. Exceptions

Upon Exception, the mode is automatically switched to MIPS I. The mode is saved in the lsb of the Exception PC (EPC). EPC[0] = 0 indicates that the Exception occurred while executing code in MIPS I mode; EPC[0] = 1 indicates that the Exception occurred in MIPS16 mode. The typical program will save the EPC to a general register and later return to the main program with a JR instruction, causing the proper ISA mode to be restored.

4.4. No Delay Slots

Consistent with the MIP16 emphasis on code density, there are no load delay or branch delay slots. In other words, the instruction following the branch is executed only if the branch is not taken. [MIPS16 *jumps* (JAL, JALX, JR, JALR) have a single delay slot, the same as in MIPS I. For jumps, the target address is always taken. Thus, there is no risk that the delay slot cannot be used to do useful work: the instruction from the target can be moved to the delay slot, if necessary.]

For MIPS16 loads, the instruction following the load can reference the loaded register (as in MIPS II). This feature is present because the MIPS I compiler is not always successful in scheduling a useful instruction in the delay slot and must occasionally resort to a NOP, reducing code density. This possibility is eliminated in MIPS16.

5. LX5280 DSP Programming Model

The LX5280 supports Lexra's Radiax DSP extensions to the MIPS-1 instruction set. This chapter describes the Radiax extensions in detail. Section 5.1 describes each of the Radiax instructions. Section 5.2 describes the instruction encoding.

The following conventions are employed in the instruction descriptions.

« »	Encloses a list of syntax choices, from which one must be chosen.
value[3]	Bits selected from a value.
MNEM[.OPT]	Indicates an optional form of instruction an mnemonic.

5.1. Radiax Instructions

The Radiax instruction extensions include MAC operations, vector-addressing, and enhanced extensions to the MIPS-1 ALU instructions.

5.1.1. Radiax Dual-MAC Instructions

Table 20: Radiax Dual-MAC Instructions

Instruction	Syntax and Description
Dual Move to Accumulator	MTA2[.G] rS, «mD, mDh, mDl» If MTA2, and mDh(mDl) is selected, sign-extend the contents of general register rS to 40-bits and move to accumulator register mDh(mDl). If MTA2, and mD is selected, update both mDh and mDl with the 40-bit, sign-extended contents of the same rS. If MTA2.G is selected, the accumulator register bits [39:32] are updated with rS[31:24]; bits [31:00] of the accumulator are unchanged. (The .G option is used to restore the upper-bits of the accumulator from the general register file; typically, following an exception.)
Move From Accumulator	MFA rD, «mTh, mTl» [,n] Move the contents of accumulator register mTh or accumulator register mTl to register rD with optional right shift. Bits [31+n : n] from the accumulator register are transferred to rD[31:00]. The range n = 0 - 8 is permitted for the output alignment shift amount. In the case of n = 0, the field may be omitted.
Dual Move From Accumulator	MFA2 rD, mT [,n] Move the contents of the upper halves of accumulator register pair mT to register rD with optional right shift. The rD[31:16] are taken from mTh and rD[15:00] from the corresponding mTl. mTh[31+n: 16+n] || mTl[31+n : 16+n] from the accumulator register pair are transferred to rD[31:00]. The range n = 0 - 8 is permitted for the output alignment shift amount. In the case of n = 0, the field may be omitted.
Divide	DIVA mD, rS, rT The contents of register rS is divided by rT, treating the operands as signed 2's complement values. The remainder is sign-extended to 40-bits and stored in mDh and the quotient is sign-extended to 40-bits and stored in mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO.
Divide Unsigned	DIVAU mD, rS, rT The contents of register rS is divided by rT, treating the operands as unsigned values. The remainder is zero-extended to 40-bits and stored in mDh and the quotient is zero-extended to 40-bits and stored in mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO.

Instruction	Syntax and Description
Multiply (32-bit)	MULTA mD, rS, rT The contents of register rS is multiplied by rT, treating the operands as signed 2's complement values. The upper 32-bits of the 64-bit product is sign-extended to 40-bits and stored in mDh and the lower 32-bits is zero-extended to 40-bits and stored in the corresponding mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the product is left shifted by one bit, and furthermore, if both operands are -1 then the product is set to positive signed, all ones fraction, prior to the shift. If both MMD[MT] and MMD[MF] are 1, the result is undefined.
Multiply Unsigned (32-bit)	MULTAU mD, rS, rT The contents of register rS is multiplied by rT, treating the operands as unsigned values. The upper 32-bits of the 64-bit product is zero-extended to 40-bits and stored in mDh and the lower 32-bits is zero-extended to 40-bits and stored in the corresponding mDl. m0h[31:00] is also called HI. m0l[31:00] is also called LO. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the result is undefined.
Dual Multiply (16-bit)	MULTA2 «mD, mDh, mDl», rS, rT The contents of register rS is multiplied by rT, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] and the product is sign-extended to 40-bits and stored in mDh. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] and the product is sign-extended to 40-bits and stored in mDl. If the destination is mD, both operations are performed and the two products are stored in the accumulator register pair mD. If MMD[MF] is 1, then each product is left shifted by one bit, and furthermore, for each multiply, if both operands are -1 then the product is set to positive signed, all ones fraction.
Dual Multiply and Negate (16-bit)	MULNA2 «mD, mDh, mDl», rS, rT The contents of register rS is multiplied by rT, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] and the product is sign-extended to 40-bits, negated (i.e. subtracted from zero) and stored in mDh. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] and the product is sign-extended to 40-bits, negated (i.e. subtracted from zero) and stored in mDl. If the destination is mD, both operations are performed and the two products are stored in the accumulator register pair mD. If MMD[MF] is 1, then each product is left shifted by one bit prior to sign-extension and negation, and furthermore, for each multiply, if both operands are -1 then the product is set to positive signed, all ones fraction prior to sign-extension and negation.
Complex Multiply,	CMULTA mD, rS, rT rS[31:16] is interpreted as the real part of a complex number. rS[15:00] is interpreted as the imaginary part of the same complex number. Similarly for the contents of general register rT. As the result of CMULTA, mDh is updated with the real part of the product, sign-extended to 40-bits and mDl is updated with the imaginary part of the product, sign-extended to 40-bits. If MMD[MF] is 1, then each product is left shifted by one bit, and furthermore, for each multiply, if both operands are -1 then the product is set to positive signed, all ones fraction, prior to the addition of terms.
32-bit Multiply-Add with 72-bit accumulate	MADDA mD, rS, rT The contents of register rS is multiplied by rT treating the operands as signed 2's complement values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the product is left shifted by one bit, and furthermore, if both operands are -1 then the product is set to a positive signed, all ones fraction. If both MMD[MT] and MMD[MF] are 1, then the result of the multiply is undefined. The 64-bit product is sign-extended to 72-bits and added to the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.

Instruction	Syntax and Description
32-bit unsigned Multiply-Add with 72-bit accumulate	MADDAU mD, rS, rT The contents of register rS is multiplied by rT treating the operands as unsigned values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the result of the multiply is undefined. The 64-bit product is zero-extended to 72-bits and added to the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.
Dual Multiply-Add, optional saturation	MADDA2[.S]«mD, mDh, mDl», rS, rT The contents of register rS is multiplied by rT and added to an accumulator register, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] then sign-extended and added to mDh[39:00]. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] then sign-extended and added to mDl[39:00]. If the destination is mD, both operations are performed and the two results are stored in the accumulator register pair mD. If MADDA2.S the result of each addition is saturated before storage in the accumulator register. The multiplies are subject to MMD[MF] as in MULTA2. The saturation point is selected as either 40 or 32 bits by MMD[MS].
32-bit Multiply-Subtract with 72-bit accumulate	MSUBA mD, rS, rT The contents of register rS is multiplied by rT treating the operands as signed 2's complement values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the product is left shifted by one bit, and furthermore, if both operands are -1 then the product is set to a positive signed, all ones fraction. If both MMD[MT] and MMD[MF] are 1, then the result of the multiply is undefined. The 64-bit product is sign-extended to 72-bits and subtracted from the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.
32-bit unsigned Multiply-Subtract with 72-bit accumulate	MSUBAU mD, rS, rT The contents of register rS is multiplied by rT treating the operands as unsigned values. If MMD[MT] is 1, then the partial product rS[15:00] x rT[15:00] is not included in the total product. If MMD[MF] is 1, then the result of the multiply is undefined. The 64-bit product is zero-extended to 72-bits and subtracted from the concatenation mDh[39:0] || mDl[31:0], ignoring mDl[39:32]. The lower 32 bits of the result are zero-extended to 40-bits and stored into mDl. The upper 40-bits of the result are stored into mDh.
Dual Multiply-Sub, optional saturation	MSUBA2[.S] «mD, mDh, mDl», rS, rT The contents of register rS is multiplied by rT and subtracted from an accumulator register, treating the operands as signed 2's complement values. If the destination register is mDh, rS[31:16] is multiplied by rT[31:16] then sign-extended and subtracted from mDh[39:00]. If the destination register is mDl, rS[15:00] is multiplied by rT[15:00] then sign-extended and subtracted from mDl[39:00]. If the destination is mD, both operations are performed and both results are stored in the accumulator register pair mD. If MSUBA2.S the result of each subtraction is saturated before storage in the accumulator register.
Add Accumulators	ADDMA[.S]mD«h,l», mS«h,l», mT«h,l» The contents of accumulator mTh or mTl is added to the contents of accumulator mSh or mSl, treating both registers as signed 40-bit values. mDh or mDl is updated with the result. If ADDMA.S, the result is saturated before storage. The saturation point is selected as either 40 or 32 bits by MMD[MS].
Subtract Accumulators	*SUBMA[.S] mD«h,l», mS«h,l», mT«h,l»* The contents of accumulator mTh or mTl is subtracted from the contents of accumulator mSh or mSl, treating both registers as signed 40-bit values. mDh or mDl is updated with the result. If SUBMA.S, the result is saturated before storage.The saturation point is selected as either 40 or 32 bits by MMD[MS].

Instruction	Syntax and Description
Dual Round	RNDA2 «mT, mTh, mTl» [,n] The accumulator register mTh or mTl is rounded, then updated. If mT, the accumulator register pair mTh/mTl are each rounded, then updated. The rounding mode is selected in MMD field "RND". The least significant bit of precision in the accumulator register after rounding is: 16+n. Bits [15+n : 00] are zeroed. The range n = 0 - 8 is permitted for the output alignment shift amount. In the case of n = 0, the field may be omitted.

Nomenclature:

rS, rT	=	r0 - r31
mD	=	mDh || mDl; also for mT
mDh	=	m0h - m3h; also for mSh, mTh
mDl	=	m0l - m3l; also for mSh, mTh
HI	=	m0h[31:00]
LO	=	m0l[31:00]

5.1.2. Cycle-by-Cycle Usage for Dual MAC Instructions

The Dual MAC eliminates all programming hazards for its instructions by stalling the pipeline when necessary. It does this both to avoid resource conflicts and to wait for results of a first instruction to be ready before attempting to use those results in a second instruction. This means that there are no programming restrictions in order to obtain correct results from a sequence of Dual MAC instructions.

However, the most efficient use of the Dual MAC hardware is obtained when the program avoids these stalls. This can be done by scheduling the instructions properly. Table 52 on page 123 indicates the number of cycles that must be present between MAC instructions to avoid stalls. In addition several instruction sequences are presented that represent the most efficient use of the Dual MAC for the "inner loop" of some common DSP algorithms. Typically, these make use of the multiple accumulators in the Dual MAC.

The following code sequences indicate the most efficient use of the Dual MAC for coding the inner loop of some common DSP algorithms. The algorithms are presented for 16-bit operands with 16-bit results, as well as 32-bit operands with 32-bit results. The algorithms assume that fractional arithmetic is used. Therefore, for the 32-bit results of a 32x32 multiply, only the HI half of the target accumulator pair is retrieved or used.

In these examples, only the Dual MAC instructions are shown. The other pipe is used to fetch and store operands and take care of loop housekeeping functions. The loops may need to be unrolled to take full advantage of the multiple Dual MAC accumulators.

CASE 1: 16-bit inner product. SUM = SUM + Ai*Bi

Assuming packed operands, two multiply-adds per cycle:

```
MADDA2 m0,r1,r2
MADDA2 m0,r3,r4
MADDA2 m0,r5,r6
MADDA2 m0,r7,r8
...
```

CASE 2: 16-bit vector product loop. Ci = Ai*Bi

Assuming packed fractional operands, two multiplies per two cycles using two accumulator pairs.

```
MULTA2 m0,r1,r2
MFA2   m1,r8
MULTA2 m1,r3,r4
MFA2   m0,r7
...
```

CASE 3: 16-bit complex vector product. Ci = Ai *complex Bi

Assuming fractional operands packed as 16-bit real, 16-bit imaginary. One complex multiply every two cycles using two accumulator pairs.

```
CMULTA m0,r1,r2
MFA2   m1,r8
CMULTA m1,r3,r4
MFA2   m0,r7
...
```

CASE 4: 32-bit inner product loop. SUM = SUM + Ai*Bi

Achieves a multiply-accumulate every other cycle using one accumulator.

```
MADDA m0,  r1,  r2
non-DualMAC op
MADDA m0,  r3,  r4
non-DualMAC op
...
```

CASE 5: 32-bit vector product loop. Ci = Ai*Bi

Assuming fractional 32-bit operands so that the MFA waits for the HI result of the MULTA. Achieves one multiply per two cycles using all the accumulators.

```
MULTA m0, r1, r2
MFA   r9, m1h
MULTA m1, r3, r4
MFA   r10,m2h
MULTA m2, r5, r6
MFA   r11,m3h
MULTA m3, r7, r8
MFA   r12,m0h
...
```

CASE 6: 32-bit complex vector product. Ci = Ai *complex Bi

Assuming fractional 32-bit operands so that the ADDMA/SUBMA waits for the HI result of the second MULTA. Achieves one complex multiply per ten cycles using all the accumulators, with two inserted instructions. This is a good example of the cycles needed from MULTA to SUBMA/ADDMA (5 cycles for HI) and from SUBMA/ADDMA to MFA (2 cycles).

```
MULTA m0, r1, r4  ; a[2i]   * b[2i+1]
MFA   rimag, m1h
MULTA m1, r2, r3  ; a[2i+1] * b[2i]
SUBMA m3h,m2h,m3h ; c[2i-2] = a[2i-2]*b[2i-2] - a[2i-1]*b[2i-1]
non-DualMAC op
MULTA m2, r1, r3  ; a[2i]   * b[2i]
MFA   rreal, m3h
```

```
MULTA m3, r2, r4  ; a[2i+1] * b[2i+1]
ADDMA m1h,m0h,m1h ; c[2i+1] = a[2i+1]*b[2i] + a[2i]*b[2i+1]
non-DualMAC op
...
```

5.1.3. Vector Addressing Instructions

Table 21: Vector Addressing Instructions

Instruction	Syntax and Description
Load Twinword	LT rT, displacement(base) The displacement, in bytes, is a signed 14-bit quantity that must be divisible by 8 (since it occupies only 11 bits of the instruction word). Sign-extend the displacement to 32-bits and add to the contents of register *base* to form the address *temp*. Load contents of word addressed by *temp* into register rT (which must be an even register). Load contents of word addressed by *temp*+4 into register rT+1.
Store Twinword	ST rT, displacement(base) The displacement, in bytes, is a signed 14-bit quantity that must be divisible by 8 (since it occupies only 11 bits of the instruction word). Sign-extend the displacement to 32-bits and add to the contents of register *base* to form the address *temp*. Store contents of register rT (which must be an even register) into word addressed by *temp*. Store contents of register rT+1 into word addressed by *temp*+4.
Load Twinword, Pointer Increment, optional circular buffer	LTP[.Cn] rT, (pointer)stride Let *temp* = contents of register *pointer*. Load contents of word addressed by *temp* into register rT (which must be an even register). Load contents of word addressed by *temp*+4 into register rT+1. The stride, in bytes, is a signed 11-bit quantity that must be divisible by 8 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Word, Pointer Increment, optional circular buffer	LWP[.Cn] rT, (pointer)stride Load contents of word addressed by register *pointer* into register rT. The stride, in bytes, is a signed 10-bit quantity that must be divisible by 4 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address.Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Halfword, Pointer Increment, optional circular buffer	LHP[.Cn] rT, (pointer)stride Load contents of sign-extended halfword addressed by register *pointer* into register rT. The stride, in bytes, is a signed 9-bit quantity that must be divisible by 2 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Halfword Unsigned, Pointer Increment, optional circular buffer	LHPU[.Cn] rT, (pointer)stride Load contents of zero-extended halfword addressed by register *pointer* into register rT. The stride, in bytes, is a signed 9-bit quantity that must be divisible by 2 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Load Byte, Pointer Increment, optional circular buffer	LBP[.Cn] rT, (pointer)stride Load contents of sign-extended byte addressed by register *pointer* into register rT. The stride, in bytes, is a signed 8-bit quantity. Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.

Instruction	Syntax and Description
Load Byte Unsigned, Pointer Increment, optional circular buffer	LBPU[.Cn] rT, (pointer)stride Load contents of zero-extended byte addressed by register *pointer* into register rT. The stride, in bytes, is a signed 8-bit quantity. Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Twinword, Pointer Increment, optional circular buffer	STP[.Cn] rT, (pointer)stride Let *temp* = contents of register *pointer*. Store contents of register rT (which must be an even register) into word addressed by *temp*. Store contents of register rT+1 into word addressed by *temp*+4. The stride, in bytes, is a signed 11-bit quantity that must be divisible by 8 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address.Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Word, Pointer Increment, optional circular buffer	SWP[.Cn] rT, (pointer)stride Store contents of register rT into word addressed by register *pointer*. The stride, in bytes, is a signed 10-bit quantity that must be divisible by 4 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Halfword, Pointer Increment, optional circular buffer	SHP[.Cn] rT, (pointer)stride Store contents of register rT[15:00] into 16-bit halfword addressed by register *pointer*. The stride, in bytes, is a signed 9-bit quantity that must be divisible by 2 (since it occupies only 8 bits of the instruction word). Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Store Byte, Pointer Increment, optional circular buffer	SBP[.Cn] rT, (pointer)stride Store contents of register rT[07:00] into byte addressed by register *pointer*. The stride, in bytes, is a signed 8-bit quantity. Sign-extend the stride to 32-bits and add to contents of register *pointer* to form next address. Update *pointer* with the calculated next address. ".Cn" selects circular buffer n = 0 - 2. See Note 2.
Move To Radiax, User	MTRU rT, RADREG Move the contents of register rT to one of the User Radiax registers: cbs0 - cbs2, cbe0 - cbe2, mmd, lpc0, lpe0, lps0. This instruction has a single delay slot before the MMD register takes effect (all other registers have no delay slot).
Move From Radiax, User	MFRU rT, RADREG Move the contents of the designated User Radiax register (cbs0 - cbs2, cbe0 - cbe2, mmd, lpc0, lps0, lpe0) to register rT.

Nomenclature:

rT	=	r0 - r31, and must be even for LT, ST, LTP[.Cn], STP[.Cn]
base, pointer	=	r0 - r31
stride	=	8/9/10/11-bit signed value (in bytes) for byte/halfword/ word/twinword ops.
displacement	=	14-bit signed value, in bytes
RADREG	=	cbs0 - cbs2, cbe0 - cbe2, mmd, lpc0, lps0, lpe0

Notes:

1. For LTP[.Cn], LWP[.Cn], LHP(U)[.Cn], LBP(U)[.Cn], rT = *pointer* is unsupported.
2. When a circular buffer is selected, the update of the pointer register is performed according to the following algorithm, which depends on the sign of the stride and the granularity of the access. A stride exactly equal to 0 is not supported:

LBP(U).Cn and SBP.Cn

```
if (stride > 0 && pointer[2:0] == 111 && pointer[31:3] == CBEn)
      tpointer <= { CBSn[31:3], 3'b000 }
else if (stride < 0 && pointer[2:0] == 000 && pointer[31:3] == CBSn)
      pointer <= { CBEn[31:3], 3'b111 }
else
     pointer <= pointer + stride.
```

LHP(U).Cn and SHP.Cn

```
if (stride > 0 && pointer[2:0] == 11x && pointer[31:3] == CBEn)
      pointer <= { CBSn[31:3],  3'b000 }
else if (stride < 0 && pointer[2:0] == 00x && pointer[31:3] == CBSn)
      pointer <= { CBEn[31:3] 3'b110 }
else
     pointer <= pointer + stride.
```

LWP.Cn and SWP.Cn

```
if (stride > 0 && pointer[2:0] == 1xx && pointer[31:3] == CBEn)
      pointer <= { CBSn[31:3] 3'b 000 }
else if (stride < 0 && pointer[2:0] == 0xx && pointer[31:3] == CBSn)
      pointer <= { CBEn[31:3], 3'b100 }
else
      pointer <= pointer + stride.
```

LTP.Cn and STP.Cn

```
if (stride > 0 && pointer[31:3] == CBEn)
      pointer <= { CBSn[31:3], 3'b000 }
else if (stride < 0 && pointer[31:3] == CBSn)
      pointer <= { CBEn[31:3], 3'b000 }
else
     pointer <= pointer + stride.
```

5.1.4. Radiax ALU Operations

The Radiax ALU operations include both dual 16-bit and saturating versions of the MIPS-1 ALU operations and several new ALU operations which are useful for common DSP algorithms.

Table 22: Radiax ALU Operations

Instruction	Syntax and Description
Dual Shift Left Logical Variable	SLLV2 rD, rT, rS The contents of rT[31:16] and the contents of rT[15:00] are independently shifted left by the number of bits specified by the low order four bits of the contents of general register rS, inserting zeros into the low order bits of rT[31:16] and rT[15:00]. For SLLV2, the high and low results are concatenated and placed in register rD. (Note that a [.S] option is not provided because this is a *logical* rather than *arithmetic* shift and thus the concept of arithmetic overflow is not relevant.)

Instruction	Syntax and Description
Dual Shift Right Logical Variable	SRLV2 rD, rT, rS The contents of rT[31:16] and the contents of rT[15:00] are independently shifted right by the number of bits specified by the low order four bits of the contents of general register rS, inserting zeros into the high order bits of rT[31:16] and rT[15:00]. The high and low results are concatenated and placed in register rD. (Note that a [.S] option is not provided because this is a *logical* rather than *arithmetic* shift and thus the concept of arithmetic overflow is not relevant.)
Dual Shift Right Arithmetic Variable	SRAV2 rD, rT, rS The contents of rT[31:16] and the contents of rT[15:00] are independently shifted right by the number of bits specified by the low order four bits of the contents of general register rS, sign-extending the high order bits of rT[31:16] and rT[15:00]. The high and low results are concatenated and placed in register rD. (Note that a [.S] option is not provided because arithmetic overflow/underflow is not possible.)
Add, optional saturation	ADDR[.S] rD, rS, rT 32-bit addition. Considering both quantities as signed 32-bit integers, add the contents of register rS to rT. For ADDR, the result is placed in register rD, ignoring any overflow or underflow. For ADDR.S, the result is saturated to 0 || 1^{31} (if overflow) or 1 || 0^{31} (if underflow) then placed in rD. ADDR[.S] will not cause an Overflow Trap.
Dual Add, optional saturation	ADDR2[.S] rD, rS, rT Dual 16-bit addition. Considering all quantities as signed 16-bit integers, add the contents of register rS[15:00] to rT[15:00] and, independently add the contents of register rS[31:16] to rT[31:16]. For ADDR2, the high and low results are concatenated and placed in register rD ignoring any overflow or underflow. For ADDR2.S, the two results are independently saturated to 0 || 1^{15} (if overflow) or 1 || 0^{15} (if underflow) then placed in rD. ADDR2[.S] will not cause an Overflow Trap.
Subtract, optional saturation	SUBR[.S] rD, rS, rT 32-bit subtraction. Considering both quantities as signed 32-bit integers, subtract the contents of register rT from the contents of register rS. For SUBR, the result is placed in register rD ignoring any overflow or underflow. For SUBR.S, the result is saturated to 0 || 1^{31} (if overflow) or 1 || 0^{31} (if underflow) then placed in rD. SUBR[.S] will not cause an Overflow Trap.
Dual Subtract, optional saturation	SUBR2[.S] rD, rS, rT Dual 16-bit subtraction. Considering all quantities as signed 16-bit integers, subtract the contents of register rT[15:00] from rS[15:00] and, independently subtract the contents of register rT[31:16] from rS[31:16]. For SUBR2, the high and low results are concatenated and placed in register rD ignoring any overflow or underflow. For SUBR2.S, the two results are independently saturated to 0 || 1^{15} (if overflow) or 1 || 0^{15} (if underflow) then placed in rD. SUBR2[.S] will not cause an Overflow Trap.
Dual Set On Less Than	SLTR2 rD, rS, rT Dual 16-bit comparison. Considering both quantities as signed 16-bit integers, if rS[15:00] is less than rT[15:00] then set rD[15:00] to 0^{15} || 1, else to zero. Independently, considering both quantities as signed 16-bit integers, if rS[31:16] is less than rT[31:16] then set rD[31:16] to 0^{15} || 1, else to zero.
Minimum	MIN rD, rS, rT The contents of the general register rT are compared with rS considering both quantities as signed 32-bit integers. If rS < rT or rS = rT, rS is placed into rD. If, rS > rT, rT is placed into rD.
Dual Minimum	MIN2 rD, rS, rT The contents of rT[31:16] are compared with rS[31:16] considering both quantities as signed 16-bit integers. If rS[31:16] < rT[31:16] or rS[31:16] = rT[31:16], rS[31:16] is placed into rD[31:16]. If, rS[31:16] > rT[31:16], rT[31:16] is placed into rD[31:16]. A similar, independent operation is performed on rT[15:00] and rS[15:00] to determine rD[15:00].

Instruction	Syntax and Description
Maximum	MAX rD, rS, rT The contents of the general register rT are compared with rS considering both quantities as signed 32-bit integers. If rS > rT or rS = rT, rS is placed into rD. If, rS < rT, rT is placed into rD.
Dual Maximum	MAX2 rD, rS, rT The contents of rT[31:16] are compared with rS[31:16] considering both quantities as signed 16-bit integers. If rS[31:16] > rT[31:16] or rS[31:16] = rT[31:16], rS[31:16] is placed into rD[31:16]. If, rS[31:16] < rT[31:16], rT[31:16] is placed into rD[31:16]. A similar, independent operation is performed on rT[15:00] and rS[15:00] to determine rD[15:00].
Absolute, optional saturation	ABSR[.S] rD, rT Considering rT as a signed 32-bit integer, if rT > 0, rT is placed into rD. If rT < 0, -rT is placed into rD. If ABSR.S and rT = 1 || 0^{31} (the smallest negative number) then 0 || 1^{31} (the largest positive number) is placed into rD; otherwise, if ABSR and rT = 1 || 0^{31}, rT is placed into rD.
Dual Absolute, optional saturation	ABSR2[.S] rD, rT ABS[.S] operations are performed independently on rT[31:16] and rT[15:00], considering each to be 16-bit signed integers. rD is updated with the absolute value of rT[31:16] concatenated with the absolute value of rT[15:00].
Dual Mux	MUX2«[.HH], [.HL], [.LH], [.LL]» rD, rS, rT rD[31:16] is updated with rS[31:16] for MUX2.HH or MUX2.HL. rD[31:16] is updated with rS[15:00] for MUX2.LH or MUX2.LL. rD[15:00] is updated with rT[31:16] for MUX2.HH or MUX2.LH rD[15:00] is updated with rT[15:00] for MUX2.HL or MUX2.LL
Count Leading Sign bits	CLS rD, rT The binary-encoded number of redundant sign bits of general register rT is placed into rD. If rT[31:30] = 10 or 01, rD is updated with 0. If rT = 0, or if rT = 1^{32} , rD is updated with 0^{27} || 1^{5} (decimal 31).
Bit Reverse	BITREV rD, rT, rS A bit-reversal of the contents of general register rT is performed. The result is then shifted right logically by the amount specified in the lower 5-bits of the contents of general register rS, then stored in rD.

5.1.5. Conditional Operations

The LX5280 provides conditional move instructions that reduce the need for program branches, resulting in greater program efficiency.

Table 23: Conditional Operations

Instruction	Syntax and Description
Conditional Move on Equal Zero	CMVEQZ[.H] [.L] rD, rS, rT If the general register rT is equal to 0, the general register rD is updated with rS; otherwise rD is unchanged. For [.H] if rT[31:16] is equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged. For [.L] if rT[15:00] is equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged.

Instruction	Syntax and Description
Conditional Move on Not Equal Zero	CMVNEZ[.H] [.L] rD, rS, rT If the general register rT is not equal to 0, the general register rD is updated with rS; otherwise rD is unchanged. For [.H] if rT[31:16] is not equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged. For [.L] if rT[15:00] is not equal to 0, the *full 32-bit* general register rD[31:00] is updated with rS; otherwise rD is unchanged.

Usage Note:
When combined with the SLT or SLTR2 instructions, the conditional move instructions can be used to construct a complete set of conditional move macro-operations. For example:

```
if ( r3 < r4 ) r1 <-- r2:
CMVLT     r1,r2,r3,r4    =>    SLT      AT,r3,r4
                               CMVNEZ   r1,r2,AT

if ( r3  >= r4 ) r1 <-- r2:
CMVGE     r1,r2,r3,r4    =>    SLT      AT,r3,r4
                               CMVEQZ   r1,r2,AT

if ( r3 <= r4 ) r1 <-- r2:
CMVLE     r1,r2,r3,r4    =>    SLT      AT,r4,r3
                               CMVEQZ   r1,r2,AT

if ( r3  >  r4 )  r1 <-- r2:
CMVGT     r1,r2,r3,r4    =>    SLT      AT,r4,r3
                               CMVNEZ   r1,r2,AT
```

5.2. Instruction Encoding

5.2.1. Lexra Formats

The Lexra Formats are introduced into the MIPS instruction set by designating a single I-Format as "LEXOP", then using the INST[5:0] "subop" field to permit up to 64 new Lexra opcodes. Thus the new DSP opcodes model the MIPS "special" opcodes encoded in R-Format. The diagrams below illustrate the LEXOP codes using I-Format 011_111 which is unused in the MIPS I-IV ISA.

The following principles are used to resolve potential ambiguity of encoding between the new LX5280 DSP extensions and MIPS instructions:

a. LX5280 instructions with similar operations to existing MIPS instructions, but with additional operands permitted, are programmed with new Assembler mnemonics and encoded as a LEXOP. For instance:

```
multa     m1, r1, r2   is encoded as a LEXOP instruction
mult      r1, r2       is encoded as a MIPS instruction
multa     m0, r1, r2   is encoded as a LEXOP instruction  (m0 is an alias for HI/LO)
```

b. If a MIPS instruction is "extended" with new functionality, it is programmed with new Assembler mnemonics and encoded as a LEXOP. Lexra mnemonics which end in "r" indicate general register file targets; mnemonics which end in "a" indicate accumulator register targets. This convention removes

ambiguity between the Lexra op and a similar MIPS op. For example,

```
addr    r3, r1, r2    is encoded as a LEXOP instruction
add     r3, r1, r2    is encoded as a MIPS instruction
```

The MIPS *add* and the LEXOP *addr* are both signed 32-bit additions. However, on overflow the MIPS instruction triggers the Overflow Exception, while the LEXOP does not. Alternatively the result of the LEXOP will saturate if the ".s" option is selected (addr.s).

5.3. Load/Store Formats

	31 26	25 21	20 16	15 6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**base**	**rt**	**immediate**	**Lexra SUBOP**
LT	LEXOP	base	rt-even	displacement/8	LT
ST	LEXOP	base	rt-even	displacement/8	ST
	6	5	5	10	6

	31 26	25 21	20 16	15 8	7 6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**base**	**rt**	**immediate**	**cc**	**Lexra SUBOP**
LBP[.Cn]	LEXOP	pointer	rt	stride	cc	LBP
LBPU[.Cn]	LEXOP	pointer	rt	stride	cc	LBPU
LHP[.Cn]	LEXOP	pointer	rt	stride/2	cc	LHP
LHPU[.Cn]	LEXOP	pointer	rt	stride/2	cc	LHPU
LWP[.Cn]	LEXOP	pointer	rt	stride/4	cc	LWP
LTP[.Cn]	LEXOP	pointer	rt	stride/8	cc	LTP
SBP[.Cn]	LEXOP	pointer	rt	stride	cc	SBP
SHP[.Cn]	LEXOP	pointer	rt	stride/2	cc	SHP
SWP[.Cn]	LEXOP	pointer	rt	stride/4	cc	SWP
STP[.Cn]	LEXOP	pointer	rt	stride/8	cc	STP
	6	5	5	8	2	6

base, pointer, rt	Selects general register r0 - r31.
rt-even	Selects general register even-odd pair r0/r1, r2/r3, ... r30/r31
stride	Signed 2s-complement number in bytes. Must be an integral number of halfwords/words/twinwords for the corresponding instructions.
displacement	Signed 2s-complement number in bytes. Must be an integral number of twinwords.
cc	00 = select circular buffer 0 (cbs0, cbe0) 01 = select circular buffer 1 (cbs1, cbe1) 10 = select circular buffer 2 (cbs2, cbe2) 11 = no circular buffer selected

5.3.1. Arithmetic Format

	31 26	25 21	20 16	15 11	10 9	8	7	6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**rs**	**rt**	**rd**	**hl**	**0**	**s**	**d**	**Lexra SUBOP**
ADDR[.S],ADDR2[.S]	LEXOP	rs	rt	rd	0	0	s	d	ADDR
SUBR.S, SUBR2[.S]	LEXOP	rs	rt	rd	0	0	s	d	SUBR
SLTR2	LEXOP	rs	rt	rd	0	0	0	1	SLTR
SLLV2	LEXOP	rs	rt	rd	0	0	0	1	SLLV
SRLV2	LEXOP	rs	rt	rd	0	0	0	1	SRLV
SRAV2	LEXOP	rs	rt	rd	0	0	0	1	SRAV
MIN, MIN2	LEXOP	rs	rt	rd	0	0	0	d	MIN
MAX, MAX2	LEXOP	rs	rt	rd	0	0	0	d	MAX
ABSR[.S], ABSR2[.S]	LEXOP	0	rt	rd	0	0	s	d	ABSR
MUX2.[LL,LH,HL,HH]	LEXOP	rs	rt	rd	hl	0	0	1	MUX
CLS	LEXOP	0	rt	rd	0	0	0	0	CLS
BITREV	LEXOP	rs	rt	rd	0	0	0	0	BITREV
	6	5	5	5	2	1	1	1	6

rs, rt, rd	Selects general register r0 - r31.
s	Selects saturation of result. s=1 indicates that saturation is performed.
d	d=1 indicates that dual operations on 16-bit data are performed.
hl (for MUX2)	00 = LL: rD = rs[15:00] || rt[15:00] 01 = LH: rD = rs[15:00] || rt[31:16] 10 = HL: rD = rs[31:16] || rt[15:00] 11 = HH: rD = rs[31:16] || rt[31:16]

5.3.2. MAC Format A

	31 26	25 21	20 16	15 11	10	9	8	5	6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**rs**	**rt**	**md**	**0**	**u**	**gz**	**s**	**d**	**Lexra SUBOP**
CMULTA	LEXOP	rs	rt	md	0	0	0	0	0	CMULTA
DIVA(U)	LEXOP	rs	rt	md	0	u	0	0	0	DIVA
MULTA(U)	LEXOP	rs	rt	md	0	u	1	0	0	MADDA
MULTA2	LEXOP	rs	rt	md	0	0	1	0	1	MADDA
MADDA(U)	LEXOP	rs	rt	md	0	u	0	0	0	MADDA
MADDA2[.S]	LEXOP	rs	rt	md	0	0	0	s	1	MADDA
MSUBA(U)	LEXOP	rs	rt	md	0	u	0	0	0	MSUBA
MSUBA2[.S]	LEXOP	rs	rt	md	0	0	0	s	1	MSUBA
MULNA2	LEXOP	rs	rt	md	0	0	1	0	1	MSUBA
MTA2[.G]	LEXOP	rs	rt	md	0	0	g	0	1	MTA
	6	5	5	5	1	1	1	1	1	6

base, pointer, rt	Selects general register r0 - r31.
rs, rt	Selects general register r0 - r31.
md	Selects accumulator, 0NNHL where, NN = m0 - m3
HL	00 = reserved 01 = mNl 10 = mNh 11 = mN
s	Selects saturation of result. s=1 indicates that saturation is performed.
d	d=1 indicates that dual operations on 16-bit data are performed.
gz	For MTA2, used as “guard” bit. If g=1, bits [39:32] of the accumulator (pair) are loaded and bits [31:00] are unchanged. If g=0, all 40 bits [39:00] of the accumulator (or pair) are updated. For MADDA, MSUBA, used as a “zero” bit. If z = 1, the result is added to (subtracted from) zero rather than the previous accumulator value; this performs a MULTA, MULTA2 or MULNA2. If z = 0, performs a MADDA, MSUBA, MADDA2 or MSUBA2.
u	Treat operands as unsigned values (0 = signed, 1 = unsigned)

5.3.3. MAC Format B

	31 26	25 21	20 16	15 11	10 7	6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**00000**	**mt**	**rd**	**so**	**d**	**Lexra SUBOP**
MFA, MFA2	LEXOP	000000	mt	rd	so	d	MFA
RNDA2	LEXOP	000000	mt	0	so	1	RNDA
	6	5	5	5	4	1	6

rd	Selects general register r0 - r31.
mt	Selects accumulator, 0NNHL where, NN = m0 - m3
HL	00 = reserved 01 = mNl 10 = mNh 11 = mN
d	d=1 indicates that dual operations on 16-bit data are performed.
so	Encoded (“output”) shift amount n = 0 - 8 for RNDA2, MFA, MFA2 instructions.

5.3.4. MAC Format C

	31 26	25 21	20 16	15 11	10 8	7	6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**ms**	**mt**	**md**	**000**	**s**	**0**	**Lexra SUBOP**
ADDMA[.S]	LEXOP	ms	mt	md	000	s	0	ADDMA
SUBMA[.S]	LEXOP	ms	mt	md	000	s	0	SUBMA
	6	5	5	5	3	1	1	6

mt, ms, md — Selects accumulator, 0NNHL where, NN = m0 - m3

HL — 00 = reserved
01 = mNl
10 = mNh
11 = reserved

s — Selects saturation of result. s=1 indicates that saturation is performed.

5.3.5. RADIAX MOVE Format and Lexra-Cop0 MTLXC0/MFLXC0

	31 26	25 21	20 16	15 11	10 8	7	6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**00000**	**rt**	**ru/rk**	**000**	**k**	**0**	**Lexra SUBOP**
MFRU	LEXOP	00000	rt	ru	000	0	0	MFRAD
MTRU	LEXOP	00000	rt	ru	000	0	0	MTRAD
MFRK	LEXOP	00000	rt	rk	000	1	0	MFRAD
MTRK	LEXOP	00000	rt	rk	000	1	0	MTRAD
	6	5	5	5	3	1	1	6

rt — Selects general register r0 - r31.

rk — Selects Radiax Kernel register in MFRK, MTRK instructions — currently all reserved. However, a Coprocessor Unusable Exception is taken in User mode if the Cu0 bit is 0 in the CP0 Status register when MFRK or MTRK is executed.

ru — Selects Radiax User register in MFRU, MTRU instructions.

00000	cbs0
00001	cbs1
00010	cbs2
00011	reserved
00100	cbe0
00101	cbe1
00110	cbe2
00111	reserved
01xxx	reserved
10000	lps0
10001	lpe0
10010	lpc0

1001	reserved
101xx	reserved
11000	mmd
11001	reserved
111xx	reserved

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

These are *not* LEXOP instructions. They are variants of the standard MTC0 and MFC0 instructions that allow access to the Lexra Coprocessor0 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 Coprocessor0 register:
00000	ESTATUS
00001	ECAUSE
00010	INTVEC
00011	reserved
001xx	reserved
01xxx	reserved
1xxxx	reserved

5.3.6. CMOVE Format

	31 26	25 21	20 16	15 11	10 9	8 6	5 0
Assembler Mnemonic	**LEXOP 011 111**	**rs**	**rt**	**rd**	**00**	**cond**	**Lexra SUBOP**
CMVEQZ[.H][.L]	LEXOP	rs	rt	rd	00	cond	CMOVE
CMVNEZ[.H][.L]	LEXOP	rs	rt	rd	00	cond	CMOVE
	6	5	5	5	2	3	6

rs, rt, rd — Selects general register r0 - r31.

cond — Condition code for rT operand referenced by the conditional move.

000 EQZ
001 NEZ
010 EQZ.H
011 NEZ.H
100 EQZ.L
101 NEZ.L
11x reserved

5.3.7. Lexra SUBOP Bit Encodings

Inst[5:3] \ Inst[2:0]	0	1	2	3	4	5	6	7
0		CMOVE			SLLV		SRLV	SRAV
1					BITREV	MUX	CLS	ABSR
2			MADDA	MSUBA			RNDA	
3			DIVA	CMULTA	MFA	MTA	ADDMA	SUBMA
4		ADDR		SUBR	MFRAD	MTRAD		
5	MIN	MAX	SLTR					
6	LBP	LHP	LTP*	LWP	LBPU	LHPU	LT*	
7	SBP	SHP	STP*	SWP			ST*	

* Indicates instructions which are implemented only in the LX5280, and not the LX5180 product.

6. Integer Multiply-Divide-Accumulate

The integer Multiply-Divide-Accumulate instructions, which are optional on other Lexra processors, are a standard feature of the LX5280 processor.

6.1. Summary of Instructions

Table 24 provides a summary of the integer Multiply-Divide-Accumulate instructions.

Table 24: Summary of MAC-DIV Instructions.

Mnemonic	Operation	Description
MTHI	HI <- Rs	pre-load accumulator, or restore saved HI
MTLO	LO <- Rs	pre-load accumulator, or restore saved LO
MFHI	Rd <- HI	read accumulator, or part of 64 bit result
MFLO	Rd <- LO	read accumulator, or part of 64 bit result
MULT	{HI,LO} <- Rs * Rt	32x32 signed multiply 64bit result
MULTU	{HI,LO} <- Rs * Rt	32x32 unsigned multiply, 64bit result
MAD	{HI,LO}<- {HI,LO} + (Rs * Rt)	32x32 signed multiply, with 64bit signed add to accum
MADU	{HI,LO}<- {HI,LO} + (Rs * Rt)	32x32 unsigned multiply, with 64bit unsigned add to accum
MSUB	{HI,LO}<- {HI,LO} - (Rs * Rt)	32x32 signed multiply, with 64bit signed add to accum
MSUBU	{HI,LO}<- {HI,LO} - (Rs * Rt)	32x32 unsigned multiply, with 64bit unsigned add to accum
MADH	HI <- HI + (Rs[15:0] * Rt[15:0])	16x16 signed multiply, with 32 bit signed add to accum
MADL	LO <- LO + (Rs[15:0] * Rt[15:0])	16x16 signed multiply, with 32 bit signed add to accum
MAZH	HI <- 0 + (Rs[15:0] * Rt[15:0])	16x16 signed multiply, add to pre-zeroed 32bit accum
MAZL	LO <- 0 + (Rs[15:0] * Rt[15:0])	16x16 signed multiply, add to pre-zeroed 32bit accum
MSBH	HI <- HI - (Rs[15:0] * Rt[15:0])	16x16 signed multiply, with 32 bit signed sub from accum
MSBL	LO <- LO - (Rs[15:0] * Rt[15:0])	16x16 signed multiply, with 32 bit signed sub from accum
MSZH	HI <- 0 - (Rs[15:0] * Rt[15:0])	16x16 signed multiply, sub from pre-zeroed 32bit accum
MSZL	LO <- 0 - (Rs[15:0] * Rt[15:0])	16x16 signed multiply, sub from pre-zeroed 32bit accum
DIV	HI <- Rs%Rt; LO <- Rs/Rt	32 by 32 signed divide with remainder
DIVU	HI <- Rs%Rt; LO <- Rs/Rt	32 by 32 unsigned divide with remainder

The processor may stall if a new MAC instruction is executed while a prior MAC operation is pending. Table 52 on page 123 indicates the number of cycles that must be present between MAC instructions to avoid stalls.

6.2. MAC-DIV Instruction Overview

- All ops except Move-to-accumulator and 32-bit multiply-accumulate functions are supported in M16 mode as well as M32 for best code compression.
- Independent 32-bit HI and LO accumulators for 16-bit Multiply-accumulate allow optimal performance in the FIR filter, or other applications which allow generation of a new result while the previous result is pending.
- Multiply-subtract instructions eliminate the need to negate coefficients.
- In case of resource conflicts, hardware manages all hazards simplifying software debug.
- There are no coding restrictions.

6.3. Op-codes for Standard Mode (32-Bit) MAC Instructions

	31 26	25 21	20 16	15 6	5 0
Mnemonic	**Major Op**	**Base**	**Rt**	**Immediate**	**Subop**
MFHI	000000	Rs	Rt	0000000000	010000
MTHI	000000	Rs	Rt	0000000000	010001
MFLO	000000	Rs	Rt	0000000000	010010
MTLO	000000	Rs	Rt	0000000000	010011
MULT	000000	Rs	Rt	0000000000	011000
MULTU	000000	Rs	Rt	0000000000	011001
MAD	011100	Rs	Rt	0000000000	00000
MADU	011100	Rs	Rt	0000000000	000001
MSUB	011100	Rs	Rt	0000000000	000100
MSUBU	011100	Rs	Rt	0000000000	000101
DIV	000000	Rs	Rt	0000000000	011010
DIVU	000000	Rs	Rt	0000000000	011011
MADH	111100	Rs	Rt	0000000000	000000
MADL	111100	Rs	Rt	0000000000	000010
MAZH	111100	Rs	Rt	0000000000	000100
MAZL	111100	Rs	Rt	0000000000	000110
MSBH	111100	Rs	Rt	0000000000	010000
MSBL	111100	Rs	Rt	0000000000	010010
MSZH	111100	Rs	Rt	0000000000	010100
MSZL	111100	Rs	Rt	0000000000	010110
	6	5	5	10	6

6.4. Op-codes for MIPS-16 (16-Bit) Mode MAC Instructions

	15 11	10 8	7 5	4 0
Mnemonic	**major op**	**base**	**rt**	**subop**
MFHI	11101	rx	ry	10000
MTHI	not supported by MIPS-16 architecture			
MFLO	11101	rx	ry	10010
MTLO	not supported by MIPS-16 architecture			
MULT	11101	rx	ry	11000
MULTU	11101	rx	ry	11001
MAD	not supported by MIPS-16 architecture			
MADU	not supported by MIPS-16 architecture			
MSUB	not supported by MIPS-16 architecture			
MSUBU	not supported by MIPS-16 architecture			
DIV	11101	rx	ry	11010
DIVU	11101	rx	ry	11011
MADH	11111	rx	ry	00000
MADL	11111	rx	ry	00010
MAZH	11111	rx	ry	00100
MAZL	11111	rx	ry	00110
MSBH	11111	rx	ry	10000
MSBL	11111	rx	ry	10010
MSZH	11111	rx	ry	10100
MSZL	11111	rx	ry	10110
	5	3	3	5

6.5. Non-Standard Instruction Descriptions

Table 25: 16-bit Multiply and Multiply-Accumulate Instructions

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

Table 26: 32-Bit Multiply-Accumulate Instructions

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

Notes:

The 32-bit op-codes are unchanged (from the MIPS-I standard) for the existing MULT, DIV, MF, and MT instructions. The MAD, MADU, MSUB, and MSUBU are new Special2 opcodes, also standard to several processors. In M32 mode, the new instructions are all R-format with bits 31:26 = 6'b111100. Bits 5:0 determine the specific operation, as shown. In M16 mode, the new instructions are all RR-format with bits 15:11 = 5'b11111. Bits 4:0 determine the specific operation, as shown in Section 6.4.

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

The MxxH and MxxL instructions can be freely interleaved. That is, adds and subtracts from either accumulator can be combined in a sequence with the two accumulators functioning "in parallel."

The MxZx instructions can be used as stand-alone 16-bit signed multiply. This removes the need for a "MTHI, zero" instruction at the beginning of a multiply-accumulate sequence, for example:

```
MAZH r1,r2
MADH r3,r4
MADH r5,r6
MADH r7,r8
any op that doesn't write HI
any op that doesn't write HI
MFHI r9
```

In the above sequence, the two non-HI ops are not necessary for correct operation but the pipeline will stall if they are not used, so it is more efficient to perform useful work in those slots.

For the MULTx, MADx or MSUBx instructions, the most efficient use is:

```
MULTx r1,r2
MADx  r3,r4
MSUBx r5,r6
any op that doesn't write HI or LO
any op that doesn't write HI or LO
any op that doesn't write HI or LO
MFLO  r7     /* LO or HI is available this cycle*/
MFHI  r8
```

6.6. Accessing HI and LO after multiply instructions

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

During the E cycle of any multiply operation, the initial operands are re-coded and loaded into the MANDHW and MIERHW (MBOOTH) registers. For the MULTx operations, the multiply cycles can be labeled M1 through M3. Then the following timing diagram is valid:

```
    MULTx    I  S  E  M1 M2 M3
    LO/HI valid              X
    any op      I  S  E  M  W
    any op         I  S  E  M  W
    MFLO              I  S  E  M  W
or  MFHI              I  S  E  M  W
```

For the MADx operations, the pipeline cycles after E can be labeled as C (carry save), and A (accumulate). Then the following timing diagram is valid:

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

6.7. Divider Overview and Register Usage

Given a dividend DEND, and divisor DVSR, the divider generates a quotient QUOT and remainder REM that satisfy the following conditions, regardless of the signs of DEND and DVSR:

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

where REM and DEND have the same sign.

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

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

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

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

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

7. LX5280 Local Memory

7.1. Local Memory Overview

This chapter describes how memories are configured and connected to the LX5280 using the Local Memory Interfaces (LMIs). This section provides a brief summary of the conventions and supported memories. Section 7.2 describes the control register that allows software control over certain aspects of the LMIs. The subsequent sections cover each of the LMIs in detail.

This chapter also discusses configuration options and the ports that customers must access to connect application specific RAM and ROM devices that are used by the LX5280 LMIs. All of the signals between the processor core, the LMIs, RAMs and the system bus controller are automatically configured by *lconfig*, the LX5280 configuration tool. *Lconfig* also produces documentation of the exact RAMs required for the chosen configuration settings, and writes RAM models used for RTL simulation.

The LMIs connect to RAMs that service the LX5280 processor's local instruction and data busses. The LMIs also provide the pathways from the processor to the system bus. The LX5280 includes an LMI for each of the local memory types. The sizes of the RAMs and ROMs are customer selectable. The LX5280 LMIs directly support synchronous RAMs that register the address, write data, and control signals at the RAM inputs. The LMIs also supply redundant read enable and chip select lines for each RAM, which may be required for some RAM types. ROMs may also be connected, but may require a customer supplied address register at the address inputs.

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

The LX5280 is configurable for a 16, 32, 64, or 128-byte cache line size. The tag store RAM sizes shown in the tables of this chapter assume a 16-byte line size. The documentation produced by *lconfig* indicates the required tag RAMs for the selected configuration options, including the line size. As a general rule, a doubling of the line size results in halving the tag store depth.

The valid bits within tag stores are automatically cleared by the LMIs upon reset. The data cache implements a write-through protocol. Caches do not snoop the system bus. The LX5280 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 27 summarizes the LMIs that can be integrated on the local busses.

Table 27: Local Memory Interface Modules

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

7.2. Cache Control Register: CCTL

CCTL. CP0 General Register Address = 20

31-8	7	6	5	4	3-2	1	0
Reserved	IROMOff	IROMOn	IMEMOff	IMEMFill	ILock	IInval	DInval

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

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

The IROMOn and IROMOff bits of the CCTL register control the and use of the optional local IROM memory configured into the LX5280. When IROM is present and the LX5280 is reset, the LMI enables access to the IROM. A transition from 0 to 1 on IROMOff disables the IROM, allowing instruction references to be serviced IMEM, ICACHE or the system bus. A transition from 0 to 1 on IROMOn enables the IROM.

The IMEMFill and IMEMOff bits of the CCTL register control the contents and use of any local IMEM memory configured into the LX5280. When the LX5280 is reset, the LMI clears an internal register to indicate that the entire IMEM LMI contents are invalid. When IMEM is invalid, all cacheable fetches from the IMEM region will be serviced by the instruction cache, if an instruction cache is present.

A transition from 0 to 1 on IMEMFill causes the LMI to initiate a series of line read operations to fill the IMEM contents. The addresses used for these reads are defined by the configured BASE and TOP addresses of the IMEM, described in Section 7.4. The processor stalls while the entire IMEM contents are filled by the LMI. Thereafter, the LMI sets its internal IMEM valid bit and will service any access to the IMEM range from the local IMEM memory. The time that an IMEM fill takes to complete is the number of line reads needed to fill the IMEM range, multiplied by the latency of one line read, assuming there is no other system bus traffic.

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

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

To utilize the cache locking feature, software should execute at least one pass of critical subroutines or loops with ILock set to 10. After this has been done, ILock should be set to 11 to lock the critical code into set 1, and use set 0 for other code.

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

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

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

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

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

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

7.3. Instruction Cache (ICACHE) LMI

The ICACHE LMI supplies the interface for a direct mapped or two-way set associative instruction cache attached to the LX5280 local bus. The degree of associativity is specified through lconfig. The ICACHE LMI participates in cacheable instruction fetches, but only if the address is not claimed by the IMEM module. The configurations supported by ICACHE, and the synchronous RAMs required for each, are summarized in Table 28.

The instruction store for the two-way ICACHE consists of two 64-bit wide banks, with separate write-enable controls. The tag store consists of one RAM bank with tag and valid bits for set 0, and a second RAM for set 1 that holds the tag, valid, LRU (Least Recently Used), and lock bits. When a miss occurs in the two-way ICACHE, the LRU bit is examined to determine which element of the set to replace, with element 0 being replaced if LRU is 0, and element 1 being replaced if LRU is 1. The state of the LRU bit is then inverted. To optimize the timing of cache reads, the two-way ICACHE uses the state of the LRU bit to determine which element should be returned to the CPU. In the following cycle, the ICACHE determines if the correct element was returned. If not, the ICACHE takes an extra cycle to return the correct element to the CPU and inverts the LRU bit.

Table 28: ICACHE Configurations

Configuration	ICACHE_INST RAM	ICACHE_TAG RAM
no instruction cache	no RAM required	no RAM required
1K bytes, 2-way	2 x 64 x 64 bits	32 x 24 and 32 x 26 bits
2K bytes, 2-way	2 x 128 x 64 bits	64 x 23 and 64 x 25 bits
4K bytes, 2-way	2 x 256 x 64 bits	128 x 22 and 128 x 24 bits
8K bytes, 2-way	2 x 512 x 64 bits	256 x 21 and 256 x 23 bits
16K bytes, 2-way	2 x 1,024 x 64 bits	512 x 20 and 512 x 22 bits
32K bytes, 2-way	2 x 2,048 x 64 bits	1,024 x 19 and 1,024 x 21 bits
64K bytes, 2-way	2 x 4,096 x 64 bits	2,048 x 18 and 2,048 x 20 bits
1K bytes, direct mapped	128 x 64 bits	64 x 23 bits
2K bytes, direct mapped	256 x 64 bits	128 x22 bits
4K bytes, direct mapped	512 x 64 bits	256 x 21 bits
8K bytes, direct mapped	1,024 x 64 bits	512 x 20 bits

Configuration	ICACHE_INST RAM	ICACHE_TAG RAM
16K bytes, direct mapped	2,048 x 64 bits	1,024 x 19 bits
32K bytes, direct mapped	4,096 x 64 bits	2,048 x 18 bits
64K bytes, direct mapped	8,192 x 64 bits	4,096 x 17 bits

Table 29 lists the ICACHE signals that are connected to application specific modules. The IC_ prefix indicates signals that are driven by the ICACHE LMI module and received by the RAMs. The ICR_ prefix indicates signals that are driven by the ICACHE RAMs and received by the ICACHE LMI. Lexra supplies the Verilog module that makes all required connections to these wires. The width of the index and data lines depends upon the RAM connected to the LMI, and can be inferred from the Table 28.

Table 29: ICACHE RAM Interfaces

Signal	Description
IC_TAGINDEX	Tag and state RAM address (line).
ICR_TAGRD0	Tag and state RAM element 0 read path.
IC_TAGWR0	Tag and state RAM element 0 write path.
ICR_TAGRD1	Tag and state RAM element 1 read path.
IC_TAGWR1	Tag and state RAM element 1 write path.
IC_TAG0WE<N>	Tag 0 RAM write enable.
IC_TAG0RE<N>	Tag 0 RAM read enable.
IC_TAG0CS<N>	Tag 0 RAM chip select.
IC_TAG1WE<N>	Tag 1 RAM write enable.
IC_TAG1RE<N>	Tag 1 RAM read enable.
IC_TAG1CS<N>	Tag 1 RAM chip select.
IC_INSTINDEX	Instruction RAM address (word).
ICR_INST0RD	Instruction RAM element 0 read path.
ICR_INST1RD	Instruction RAM element 1 read path.
IC_INSTWR	Instruction RAM write path (to both elements).
IC_INST0WE<N>[1:0]	Instruction RAM 0 write enable.
IC_INST0RE<N>	Instruction RAM 0 read enable.
IC_INST0CS<N>	Instruction RAM 0 chip select.
IC_INST1WE<N>[1:0]	Instruction RAM 1 write enable.
IC_INST1RE<N>	Instruction RAM 1 read enable.
IC_INST1CS<N>	Instruction RAM 1 chip select.

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

7.4. Instruction Memory (IMEM) LMI

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

The configurations supported by IMEM, and the synchronous RAMs required for each, are summarized in Table 30.

Table 30: 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 31 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 30.

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 LX5280. The size of the memory region must be a power of two, and must be naturally aligned.

Table 31: 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.

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

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

7.5. Instruction ROM (IROM) LMI

The IROM LMI supplies the interface for an optional read-only local instruction store. The IROM serves a fixed range of the physical address space, determined by configuration settings in *lconfig*. IROM may be disabled via a hardware configuration pin, CFG_IROFF. IROM may also be enabled and disabled under software control as described in Section 7.2, Cache Control Register: CCTL. The IROM is a convenient, low-cost alternative to a cache that makes read-only instruction memory available to the core for high-speed access.

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

Table 32: IROM Configurations

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

Table 33 lists the IROM signals that are connected to application specific modules. The IR_ prefix indicates signals that are driven by the IROM LMI module and received by the ROM. The IRR_ prefix indicates signals that are driven by ROM and received by the IROM LMI. The CFG_ prefix identifies configuration ports on the IROM LMI that are typically wired to constant values. Lexra supplies the Verilog module that makes all required connections to these wires. The width of the index and data lines depends upon the ROM connected to the LMI, and can be inferred from Table 31.

The CFG_ wires define where IROM is mapped into the physical address space. This configuration information defines the local bus address region of the IROM. It also determines the address of the external resources which are accessed when an IROM miss occurs. The lconfig utility supplied by Lexra will verify that the configured address range does not interfere with other regions defined by the LX5280. Note that the

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

Table 33: IROM ROM Interfaces

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

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

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

The DCACHE LMI supplies the interface for a direct mapped, write through data cache attached to the LX5280 local bus. The DCACHE LMI participates in cacheable data reads and writes, but only if the address is not claimed by the DMEM LMI. The configurations supported by DCACHE, and the synchronous RAMs required for each, are summarized in Table 34.

The direct mapped DCACHE module services word or twin-word read requests from the core in one cycle when the request hits the cache. Byte or half-word reads that hit the data cache require an extra cycle for alignment. The data cache can stream word and twin-word reads or writes that hit the cache at the rate of one per cycle. If the LX5280 is configured to work with RAMs that have word write granularity, byte or half-word writes that follow any write by one cycle and hit the cache require an extra cycle to merge the data with the current cache contents. Alternatively, the LX5280 can be configured to work with RAMs support byte write granularity, which eliminates the extra cycle. See Appendix C, LX5280 Pipeline Stalls, for detailed descriptions of these and other pipeline stall conditions.

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

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

Table 34: DCACHE Configurations

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

Table 35 lists the DCACHE signals that are connected to application specific modules. The DC_ prefix indicates signals that are driven by the DCACHE LMI module and received by the RAMs. The DCR_ prefix indicates signals that are driven by the DCACHE RAMs and received by the DCACHE LMI. Lexra supplies the Verilog module that makes all required connections to these wires. The width of the index and data lines depends upon the RAM connected to the LMI, and can be inferred from Table 34.

Table 35: DCACHE RAM Interfaces

Signal	Description
DC_TAGINDEX	Tag and state RAM address.
DCR_TAGRD	Tag and state RAM read path.
DC_TAGWR	Tag and state RAM write path.
DC_TAGWE<N>	Tag and state RAM write enable.
DC_TAGRE<N>	Tag and state RAM read enable.
DC_TAGCS<N>	Tag and state RAM chip select.
DC_DATAINDEX	Data RAM address (word).
DCR_DATARD	Data RAM read path.
DC_DATAWR	Data RAM write path.
DC_DATAWE<N>[1:0]	Data RAM write enable.
DC_DATARE<N>	Data RAM read enable.
DC_DATACS<N>	Data RAM chip select.

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

7.7. Scratch Pad Data Memory (DMEM) LMI

The DMEM LMI supplies the interface for a scratch pad data RAM attached to the LX5280 local bus. The DMEM module services in any cacheable or uncacheable data read or write operation that falls within its configured range.

Byte or half-word reads that hit the DMEM require an extra cycle for alignment. DMEM can stream word and twin-word reads or writes that hit DMEM at the rate of one per cycle. If the LX5280 is configured to work with RAMs that have word write granularity, byte or half-word writes that follow any write by one cycle and hit DMEM require an extra cycle to merge the data with the current DMEM contents. Alternatively, the LX5280 can be configured to work with RAMs support byte write granularity, which eliminates the extra cycle. See Appendix C, LX5280 Pipeline Stalls, for detailed descriptions of these and other pipeline stall conditions. 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 DMEM configurations and the synchronous RAMs required for each are summarized in the Table 36.

Table 36: 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
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 37 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 36.

The *CFG_* wires define where DMEM is mapped into the physical address space. It is not possible for any DMEM reference to result in an operation on the system bus. The *lconfig* utility supplied by Lexra will verify that the configured address range does not interfere with other regions defined for LX5280. The size of the memory region must be a power of two, and must be naturally aligned.

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

Table 37: DMEM RAM Interfaces

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

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

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

8. LX5280 System Bus

8.1. Connecting the LX5280 to internal devices

The Lexra System Bus (LBus) is the connection between the LX5280 and other internal devices, such as system memory, USB, IEEE-1394 (Firewire), and an external bus interface. The LBC uses a protocol similar to that of the Peripheral Component Interface (PCI) bus. This is a well-known and proven architecture. Adding new devices to the Lexra Bus is straightforward and the performance approaches the highest that can be achieved without adding a great deal of complexity to the protocol.

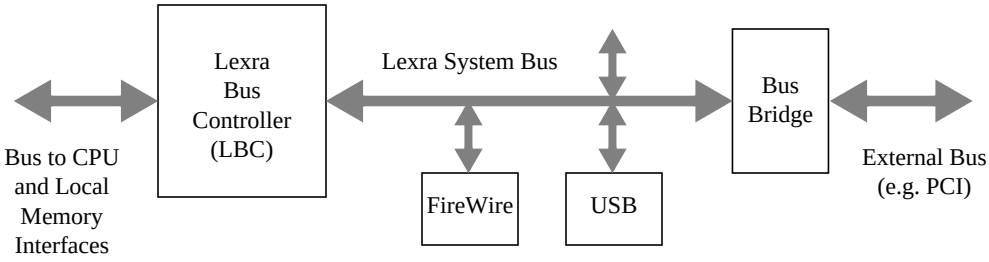


Figure 6: Lexra System Bus Diagram

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

The Lexra bus is a synchronous bus. Signals are registered and sampled at the positive edge of the bus clock. Certain logical operations may be made to the sampled signals and then new signals can be driven immediately, such as for address decoding. This allows for same-cycle turn-around. The LBC provides an optional asynchronous interface between the CPU and the Lexra bus, allowing the Lexra bus speed can be set to be any speed equal to or less than the CPU clock frequency.

The Lexra bus data path for the LX5280 is 32 bits wide. Therefore, the bus can transfer one word, halfword, or byte in one bus clock. The bus supports line and burst transfers in which several words of data are transferred. The Lexra bus accomplishes this by transferring words of data from incremental addresses on successive clock cycles.

The LBC contains a write buffer. When the CPU issues a write request to a Lexra Bus device, the address and data are saved in the buffer and sent to the device sometime later. The CPU can continue processing, having safely assumed that the write will eventually happen. This is described more thoroughly in Section 8.7.2.

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

8.2. Terminology

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

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

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

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

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

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

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

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

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

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

8.3. Bus Operations

The purpose of the Lexra bus is to connect together the various components of the system, including the LX5280 CPU, main system memory, I/O devices, and external bus bridges. Different devices have different transfer requirements. For example, the LX5280 CPU will request the bus to fetch a cache line of data from memory. I/O devices will request large blocks of data to be sent to and from memory. The Lexra bus supports the various types of transfers needed by both I/O and the processor.

The six types of bus operations are single-cycle read, line read, burst read, single-cycle write, line write (though this won't be used by the LX5280 core) and burst write.

8.3.1. Single-Cycle Read

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

8.3.2. Read Line

The read line operation reads a sequence of data from memory corresponding to the size of a cache line. The cache line size affects how many cycles are required to transfer the full line. The LX5280 and the Lexra bus support a configurable line size, specified through *lconfig*. The default line size of four words (16 bytes) is assumed here.

There are two ways that the target could transfer the data back to the initiator. The conventional way is to transfer four words of data in sequence, starting at the nearest 16-byte-aligned address smaller or equal to the address that the initiator drives. In other words, the target starts the transfer at the beginning of the line containing the requested address.

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

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

The LX5280 supports two ways of incrementing the address of a line refill. One is by *linear wrap*, where the address is simply incremented by one. The other is by *interleaved wrap*, where the next address is determined by the logical xor of the cycle count and the first word address. The interleave sequence is shown in the table below. The low-order address bits 3:2 for the first data beat are the obtained from the address of the line read request. The low order address bits for the subsequent data indicate the corresponding interleave order.

Table 38: Line Read Interleave Order

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

8.3.3. Burst Read

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

Burst read operations are used by I/O devices for block DMA transfers. The LX5280 will never issue a burst read operation.

Note that there is a difference between a 4-cycles burst and a line read. A line read may use a desired-word-first increment and wrap. A burst will always increment and will never wrap.

8.3.4. Single-Cycle Write

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

The LX5280 uses a cache with a write-through policy. All CPU instructions that write to memory generate a single-cycle write operation. (Unless the address is in the local scratchpad memory, in which case the write operation will not make it out to the Lexra bus).

8.3.5. Line Write

The line write operation is not used by the LX5280. This operation could be used by a processor that has a data cache that implements a write-back policy.

8.3.6. Burst Write

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

8.4. Signal Descriptions

Table 39: LBus Signal Description

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

8.5. LBus Commands

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

```
BCMD[6]      0=read, 1=write

BCMD[5:4]    54
             00 burst, fixed length[1]
             01 burst, unlimited number of words
             10 line, interleaved wrap[2]
             11 line, linear wrap
```

1. The number of words comes from BCMD[2:0]
2. Length is determined by the Line size, not BCMD[3:0]

```
BCMD[3:0]    3210
             1000 1 byte
             1001 2 bytes
             1010 3 bytes
             1011 1 word
             1100 2 words
             1101 reserved
             111x reserved
             0000 4 words
             0001 8 words
             0010 16 words
             0011 32 words
             01xx reserved
```

8.6. Byte Alignment

The Lexra Bus is a big endian bus. Transactions must have their data driven to the appropriate bus rails. The bus mapping is as shown in Table 40.

Table 40: LBus Byte Lane Assignment

BCMD[1:0]	ADDR[1:0]	Lexra Bus data byte lanes used			
		31:24	23:16	15:8	7:0
00	00	X			
00	01		X		
00	10			X	
00	11				X
01	00	X	X		
01	10			X	X
10	00	X	X	X	
10	01		X	X	X
11	00	X	X	X	X

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

8.7. Lexra Bus Controller

The Lexra Bus Controller (LBC) is the element of the LX5280 that connects to the Lexra Bus. It forwards all transaction requests from the LX5280 CPU to the Lexra Bus. It is an initiator and will never respond to requests from other Lexra Bus initiators.

8.7.1. LBC Commands

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

Table 41: LBus Commands Issued by the LBC

Command	BCMD[5:4]	BCMD[3:0]	Circumstances
Read Line	10 or 11, depending on configuration	0000	A cache miss during a read by the CPU
Read Single (word/halfword/byte)	00	10xx	A read by the CPU from an address in uncachable address space
Write Single (word/halfword/byte)	00	10xx	A write by the CPU into cacheable or uncachable address space

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

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

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

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

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

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

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

The write buffer bypass feature can be disabled so that reads will never pass writes.

8.7.3. LBC Read Buffer

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

When the LBC is configured with an asynchronous interface, a larger read buffer improves system and processor performance in the event of cache miss. When the LBC is configured with a synchronous interface, the cache can accept the data as fast as the LBC can read it. Therefore, there is no need for a large read buffer. Customers may reduce the size of the read buffer to a minimum size of two 32-bit entries.

In some cases, there is a need to minimize the number of gates. The read buffer size may be reduced to two or four entries for the asynchronous case. This causes a penalty in terms of Lbus utilization since now the LBC may have to de-assert IRDY if it cannot hold part of the line of data. When the read buffer is the size of a cache line, this will be relatively rare since simultaneous instruction cache and data cache misses are relatively rare. For a smaller read buffer, IRDY deassertion is almost a certainty.

8.7.4. Transfer Descriptions

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

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

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

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

4. A change in signal ownership requires one dead cycle. If an initiator gives up the bus, another initiator needs to wait for one dead cycle before it can drive the bus. If the same initiator issues a read operation and then needs to issue a write operation, it also must wait one extra cycle for the data bus to turn around.

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

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

1. The initiator gains control of the bus through arbitration (described later in this chapter).

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

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

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

5. The initiator and the target transfer data either in one cycle or in successive cycles. The agent driving data (the initiator for a write, the target for a read) indicates valid data by asserting its ready signal (IRDY or TRDY for writes and reads, respectively). The agent receiving data (target for a write, initiator for a read) indicates its ability to receive the data by asserting its ready

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

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

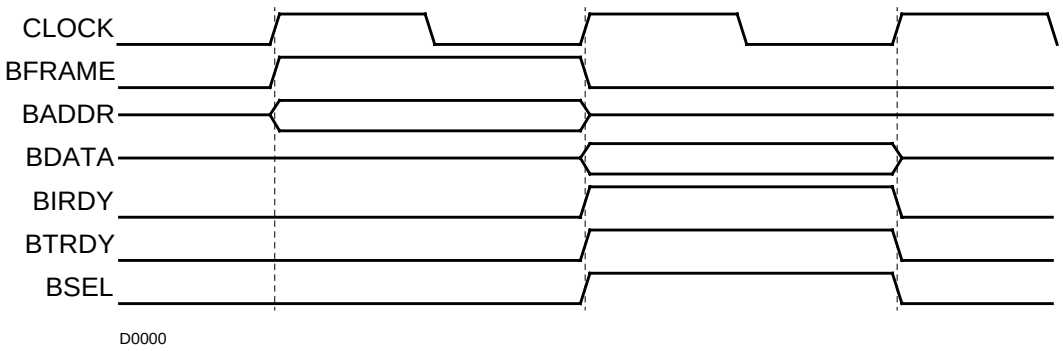
6. When the initiator is ready to send or receive the last word of data, that is, when it asserts BIRDY for the last time, it also de-asserts BFRAME. It will deassert BIRDY when the last word of data is transferred.

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

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

8.7.5. Single Cycle Read with No Waits

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



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

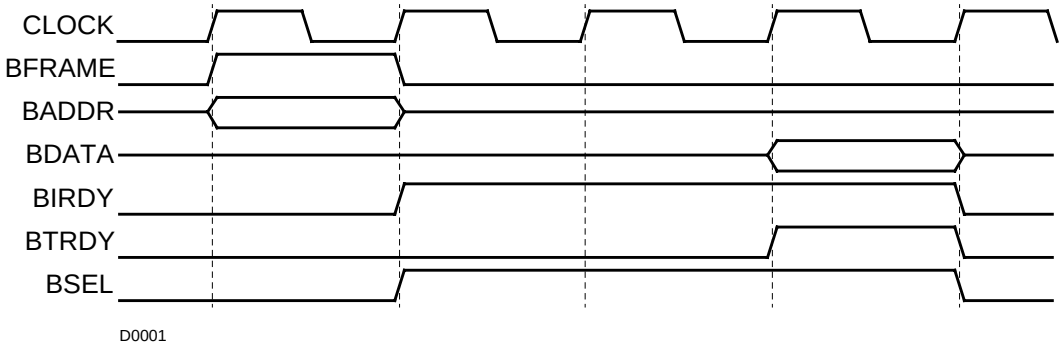
1. Initiator asserts BFRAME and drives BADDR.

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

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

8.7.6. Single Cycle Read with Target Wait

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



This is a common single-cycle read operation.

1. Initiator asserts BFRAME and drives BADDR.

2. Target asserts BSEL to indicate that it has decoded the address and is acknowledging that it is the target device. However, it is not ready to send data, so it does not assert BTRDY. Initiator de-asserts BFRAME and asserts BIRDY to indicate that the next piece of data will be the last it wants.

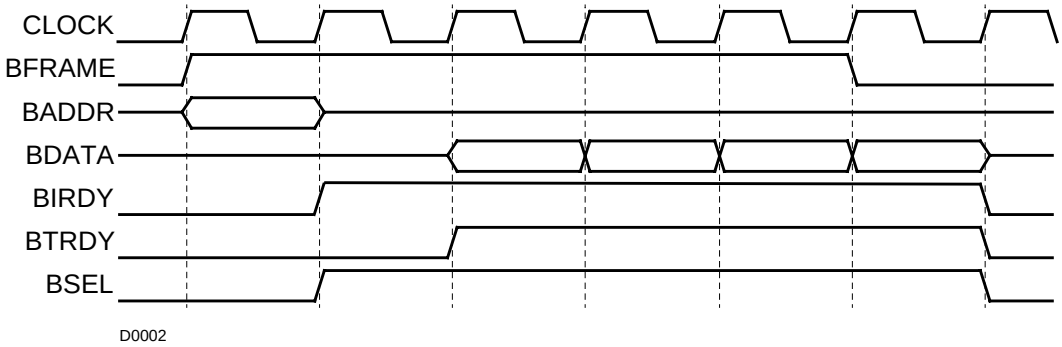
3. Target has not asserted BTRDY so no data is transferred.

4. After a second wait cycle, target drives data and asserts BTRDY to indicate that data is on the bus.

5. Target de-asserts BSEL and BTRDY. Initiator de-asserts BIRDY. Another initiator may drive the bus this cycle.

8.7.7. Line Read with No Waits

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



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

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

3. Target drives data and asserts BTRDY.

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

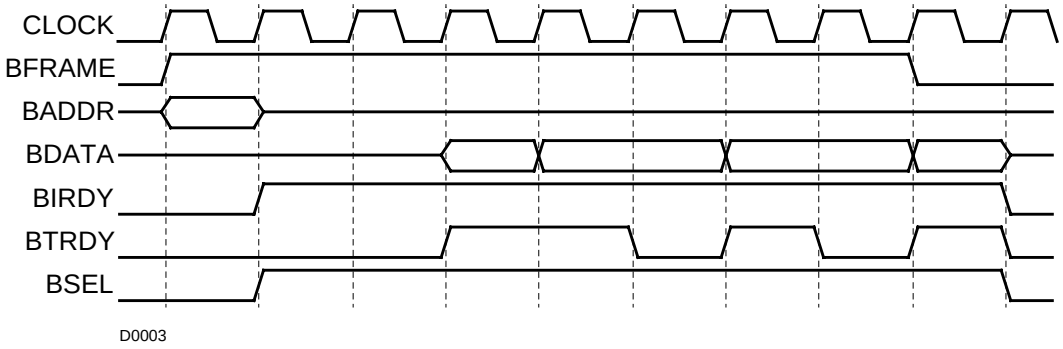
5. Target drives third word of data and continues to assert BTRDY.

6. Target drives last word of data. Initiator de-asserts BFRAME to indicate that the next word of data it receives will be the last it needs.

7. Target de-asserts BTRDY and BSEL; initiator de-asserts BIRDY. Another master may gain ownership of the bus this cycle.

8.7.8. Line Read with Target Waits

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



1. Initiator asserts BFRAME and drives BADDR.

2. Target asserts BSEL to indicate that it is acknowledging the operation. Initiator asserts BIRDY to indicate that it is ready to receive data.

3. Target waits until it has the data.

4. Target drives first word of data and asserts BTDRY.

5. Target drives second word of data and asserts BTRDY.

6. Target cannot get third word of data, so it de-asserts BTRDY.

7. Target drives third word of data and asserts BTRDY.

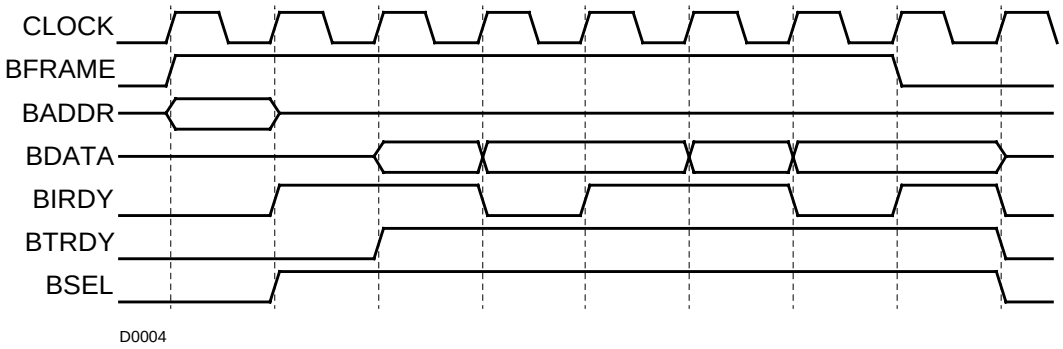
8. Target cannot get fourth word of data, so it de-asserts BTRDY.

9. Target drives fourth word of data and asserts BTRDY.

8.7.9. Line Read with Initiator Waits

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

successive cycles.



1. Initiator drives address and asserts BFRAME.
2. Target asserts BSEL. It doesn't have data, so it does not assert BTRDY. Initiator asserts BIRDY to indicate that it can accept data
3. Target now has data, so it drives the data and asserts BTRDY.
4. Target drives second word of data; initiator cannot accept it, so it de-asserts BIRDY.
5. Target holds second word of data; initiator can accept it and asserts BIRDY.
6. Target drives third word of data; initiator accepts it.
7. Target drives fourth word of data; initiator cannot accept it and de-asserts BIRDY. initiator hold BFRAME until it can assert BIRDY.
8. Initiator asserts BIRDY to accept fourth word of data. It de-asserts BFRAME to indicate this is the last word of data.

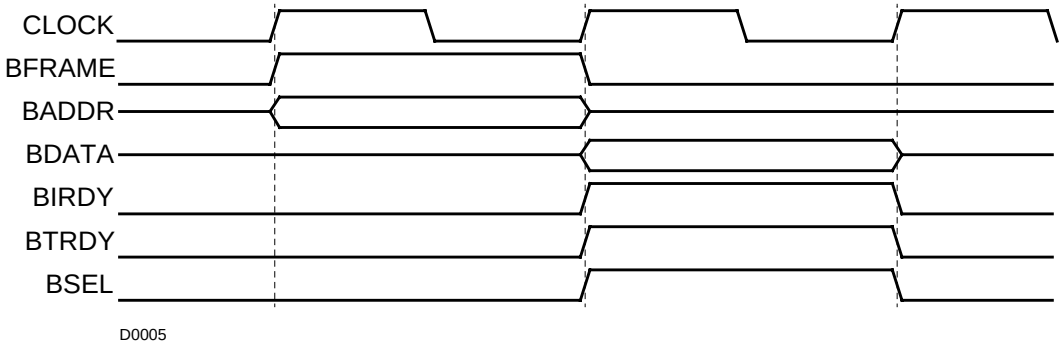
8.7.10. Burst Read

This is identical to the read line.

8.7.11. Single-Cycle Write with No Waits

A single-cycle write operation occurs almost every time the LX5280 processor executes a store instruction. This is because the cache used in the processor uses a write-through policy. Of course, writes to uncacheable address space and to an I/O device will also generate a single-word write. Single-word write operations are used to write words, halfwords and bytes.

A single-word write without waits requires two cycles.



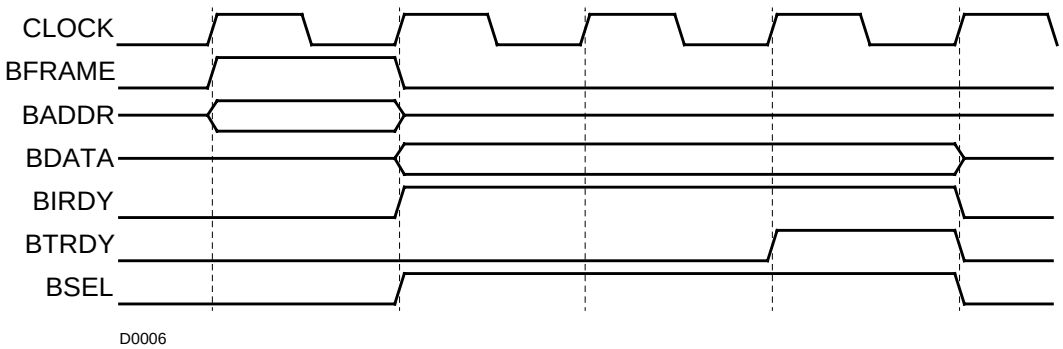
1. Initiator asserts BFRAME and drives address.

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

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

8.7.12. Single-Cycle Write with Waits

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

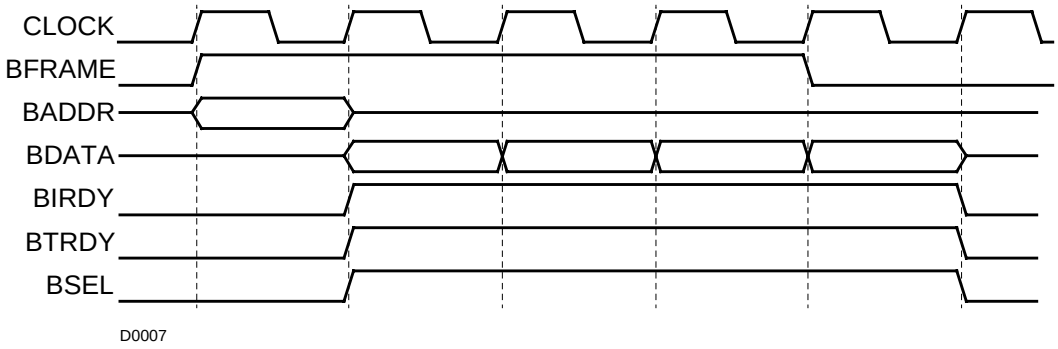


This is the same description as the above example, except that the target inserts two wait states until it asserts BIRDY to indicate acceptance of data.

8.7.13. Burst Write with No Waits

A burst write operation is generally used to transfer large amounts of data from an I/O device to memory via

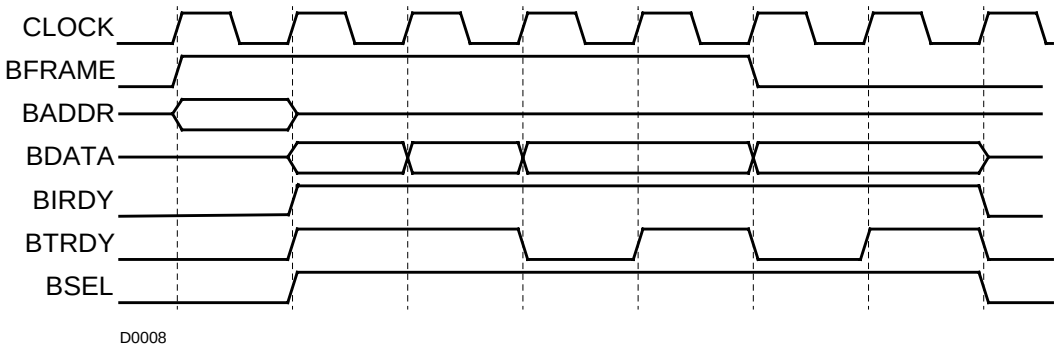
a DMA transfer. The following illustrates a best-case scenario with no wait states.



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

8.7.14. Burst Write with Target Waits

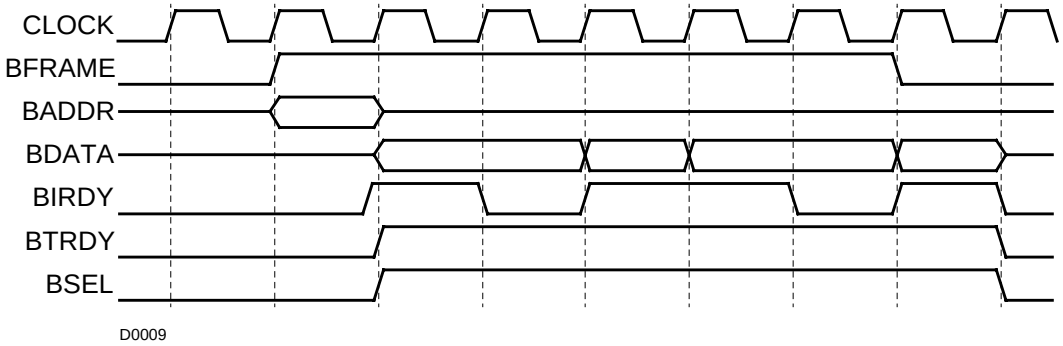
This example is similar to the above example, except that during the third and fourth data word transfer, the target cannot accept the data quickly enough, so it de-asserts BTRDY which indicates to the initiator that it should hold the data for an additional cycle.



8.7.15. Burst Write with Initiator Waits

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

waits.



8.8. LBC Signals

The table below summarizes the LX5280 LBC ports. The "LBC Port" column indicates the name of the port supplied by the LBC. The "Bus Signal" column indicates the corresponding Lexra bus signal. The LBC ports are strictly uni-directional, while the bus signals (at least conceptually) include multiple sources and sinks. The manner in which LBC ports are connected to bus signals is technology dependent, and may employ tri-state drivers or logic gating in conjunction with the LBC's LCoe, LDoe and LToe outputs.

Table 42: LBC Interface Signals

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

8.9. Arbitration

8.9.1. Rules

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

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

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

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

8.9.2. LBC behavior

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

8.10. Connecting Devices to the Bus

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

TOE is intended to qualify:

```
FRAME
IRDY
```

COE is intended to qualify:

```
CMD
ADDR
```

DOE is intended to qualify:

```
DATA
```

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

Instead of using TOE it may be desirable to instead OR all of the FRAME signals, either centrally or one OR gate for each target and master. The same holds true for IRDY, TRDY, and SEL. This simplifies the connections when a relatively few number of devices are used and there are no off-chip devices connected directly to the Lexra Bus.

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

9. LX5280 Coprocessor Interface

The LX5280 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.

9.1. 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).

9.2. 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 43: Coprocessor Interface Signals

Signal	I/O	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	I/O	Description
C<z>xcpn_M	output	Cop exception flag, one indicates exception in M stage.

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 LX5280 system clock.

9.3. 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.

9.4. 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
```

9.5. 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
```

9.5.1. 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.

9.5.2. 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.

10. LX5280 EJTAG

10.1. Introduction

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

Lexra's EJTAG solution meets all such requirements. It uses existing IEEE JTAG pins as well as fast bring-up on new designs. It provides a way of debugging all devices accessible to the processor in the same way the processor would access those devices itself. Using EJTAG, a debug probe can access all the processor internal registers and caches. It can also access devices connected to the Lexra Bus, bypassing internal caches and memories.

Software debug is enhanced by EJTAG features that allow single-stepping through code and halting on breakpoints (hardware and software, address and data with masking). For debugging problems that are artifacts of real-time interactions, EJTAG gives real-time Program Counter trace capabilities from which an accurate program execution history is derived. For the code-system perspective, PC profiling provides statistical analysis of code usage to aim code optimization.

10.2. 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 LX5280 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 LX5280 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.

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 LX5280. Information on these products is available at the following web sites.

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

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

LX5280 EJTAG implements all required features of version 2.0.0 of the EJTAG specification, and includes support for the following features:

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

10.2.1. IEEE JTAG-specific Pinout

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

Table 44: EJTAG Pinout

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

a. This pin is optional in multiprocessor configurations

Table 45: 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 46: 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

10.3. Single Processor PC Trace

The LX5280 EJTAG includes support for real-time Program Counter Trace (PC Trace). When in PC Trace

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

mode, the LX5280 will serially output a new value of the program counter whenever a change in program control occurs (i.e. branch or jump instruction, or an exception).

When the PC Trace option is set to EXPORT in lconfig, the following signals will be output from the LX5280: 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 LX5280'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.

10.3.1. 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.

10.3.2. 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 LX5280's SYSCLK, up to N PCST values are output simultaneously.

10.3.3. 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).

10.3.4. Dual Pipe PC Trace

The EJTAG PC Trace facility specifies that a PCST (PC Trace Status) code is issued if the instruction pipeline has stalled, sequentially completed an instruction, or taken an branch or jump. In order to accommodate the two pipelines in the LX5280, the capability of emitting more than one PCST code per cycle is employed. Specifically, to the external EJTAG probe, the LX5280 appears to be a single pipe machine running at twice the speed that it actually does.

Since there must be an even number of PCST codes made available at every DCLK rising edge (in the EJTAG nomenclature), the DCLK parameter "N" must be set to 2 or 4. Setting the DCLK N parameter to 2 results in DCLK running at the same frequency of SYSCLK; setting the parameter to 4 results in DCLK running at one-half the frequency of SYSCLK.

The maximum value of the N parameter is 4, and the maximum DCLK frequency is 100MHz. Therefore,

until the EJTAG specification is extended beyond N=4 or a maximum DCLK of 100MHz, the maximum SYSCLK frequency for which dual-pipe PC Trace can be used is 200 MHz.

10.3.5. Single-Processor PC Trace Pinout

Table 47: 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 48: Single-Processor PC Trace AC Characteristics[1]

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

10.3.6. 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 LX5280 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.

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

For probes that do not support a 4-bit exception code, the LX5280 can be configured via the EJTAG_XV_BITS lconfig option to use only the 3-bit standard codes. In that case, if a vectored interrupt is taken, the 3-bit code for RESET will be presented.

10.3.7. 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.

Appendix A. LX5280 Lconfig Forms

A.1. Configuration Options for the LX5280 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
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               -- 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
```

Appendix B. LX5280 Port Descriptions

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

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

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

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

The table below shows the possible port connections for the top level module of the LX5280 processor, known as lx2. The actual ports that are present depends upon *lconfig* settings. The timing information and notes have the same meaning as for the previous table.

Names that include _N indicate active low signals. All other signals are active high unless otherwise indicated.

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

Table 49: LX5280 Processor Port Summary

Port Name	I/O	Timing	Description
Clocking, Reset, Interrupts and Control			
SYSCLK	input	(3)	Processor clock.
SYSCLKF	input	(3)	Free running processor clock, if processor is configured with sleep support.
SL_SLEEPSYS_R	output	30%	Clock gating term for SYSCLK, if processor is configured with sleep support.
BUSCLK	input	(3)	Bus clock, if processor is configured with async LBC.
BUSCLKF	input	(3)	Free running bus clock, if processor is configured with async LBC sleep support.

Port Name	I/O	Timing	Description
SL_SLEEPBUS_BR	output	30%	Clock gating term for BUSCLK, if processor is configured with async LBC and sleep support.
ResetN	input	10%	Warm reset (or reset "button"), active low.
CResetN	input	10%	Cold reset (or power on), active low.
RESET_D1_R_N	input	30%	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_D1_BR_N	input	30%	BUSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_PWRON_C1_N	input	30%	Power on reset copy for JTAG.
RESET_PWRON_D1_LR_N	input	30%	SYSCLK domain power on reset for EJTAG.
RESET_D1_R_N_O	output	30%	SYSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_D1_BR_N_O	output	30%, (2)	BUSCLK domain reset combination of ResetN, CResetN, EJTAG.
RESET_PWRON_C1_N_O	output	30%	Power on reset copy for JTAG.
RESET_PWRON_D1_LR_N_O	output	30%	SYSCLK domain power on reset for EJTAG.
INTREQ_N[15:2]	input	(7)	Interrupt requests.
EXT_HALT_P	input	50%	External stall line.
EXT_SLEEPREQ_R	input	30%	External sleep request.
Configuration			
CFG_TLB_DISABLE	input	(4)	Disable TLB mappings even if tlb is present.
CFG_SLEEPENABLE	input	(4)	Sleep enable configuration.
CFG_RAD_LEXOP[5:0]	input	(4)	LEXOP encoding. Must be 011111 for LX5280.
CFG_RAD_DISABLE	input	(4)	LEXOP disable configuration. Must be one for LX5280.
CFG_SINGLEISSUE	input	(4)	Forces single instruction issue.
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.

Port Name	I/O	Timing	Description
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_EJTNMINUS1[1:0]	input	(4)	Strap with EJTAG DCLK N minus 1 configuration (0-3=1-4).
CFG_EJTMLOG2[1:0]	input	(4)	Strap with EJTAG M log2 (0-3=1,2,4,8) configuration.
CFG_EJT3BITXVTPC	input	(4)	Strap with ETJAG 3-bit TPC configuration.
CFG_EJTBIT0M16	input	(4)	Strap with EJTAG PC bit0 in TPC configuration.
CFG_DWBASE[31:10]	input	30%	Strapped with DMEM base address configuration value.
CFG_DWTOP[23:10]	input	30%	Strapped with DMEM top address configuration value.
CFG_IWBASE[31:10]	input	30%	Strapped with IMEM base address configuration value.
CFG_IWTOP[`23:10]	input	30%	Strapped with IMEM top address configuration value.
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 LX5280.
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, active low.
JTAG_RESET	input	(6)	JTAG is in TEST-LOGIC-RESET state, active low.
TAP_RESET_N_O	output	20%, (1)	TAP controller reset.
TAP_RESET_N	input	(6)	TAP controller reset.
JTAG_TDO_NR	output	50%, (1)	Test data out, active low.
JTAG_TDI	input	60%, (1)	Test data in.
JTAG_TMS	input	60%, (1)	Test mode select.
JTAG_CLOCK	input	(3)	Test clock.

Port Name	I/O	Timing	Description
JTAG_TRST_N	input	(6)	Test reset.
JTAG_CAPTURE	output	20%, (1)	JTAG is in DATA REGISTER CAPTURE state
JTAG_SCANIN	output	50%, (1)	Scan input to chain
JTAG_SCANOUT	input	50%, (1)	Scan output from chain
JTAG_IR[4:0]	output	20%, (1)	Contents of INSTRUCTION REGISTER
JTAG_SHIFT_IR	output	20%, (1)	JTAG is in SHIFT INSTRUCTION REGISTER state
JTAG_SHIFT_DR	output	20%, (1)	JTAG is in SHIFT DATA REGISTER state
JTAG_RUNTEST	output	20%, (1)	JTAG is in RUN-TEST state
JTAG_UPDATE	output	20%, (1)	JTAG is in DATA REGISTER UPDATE state
EJC_ECRPROBEEN_R	output	30%	One indicates EJTAG probe is active.
JPT_PCST_DR[M-1:0]	output	30%	EJTAG PC trace status; M= 1, 2, 4 or 8.
JPT_TPC_DR(N*3-1:0]	output	30%	EJTAG PC trace value, N= 1, 2, 3 or 4.
JPT_DCLK	output	(3)	EJTAG PC trace clock.
SEN	input	(6)	Scan enable, active high.
TMODE	input	(6)	Test mode, active high.
SIN[<k>:0]	input	(6)	Scan Input. <k> can range from 7 to 0.
SOUT[<k>:0]	output	(6)	Scan Output. <k> can range from 7 to 0.
RBC_SEL[7:0]	input	(6)	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	(6)	RAM BIST write enable, where <k> is 1 for word write granularity, 7 for byte write granularity.
RBC_RE	input	(6)	RAM BIST read enable.
RBC_CS	input	(6)	RAM BIST select.
RBC_ADDR[15:0]	input	(6)	RAM BIST address.
RBC_DATAWR[63:0]	input	(6)	RAM BIST write data.
RBM_DATARD[63:0]	output	(6)	RAM BIST read data.

Port Name	I/O	Timing	Description
LBC Interface (to LBus)			
LAddrO[31:0]	output	(2), 20%	Address.
LCmdO[6:0]	output	(2), 20%	LBC command.
LDataO[31:0]	output	(2), 20%	LBC data.
LDataI[31:0]	input	(2), 50%	System data.
LIrdyO	output	(2), 20%	LBC initiator ready.
LIrdyI	input	(2), 30%	System initiator ready.
LFrameO	output	(2), 20%	LBC transaction frame.
LFrameI	input	(2), 30%	System transaction frame.
LSel	input	(2), 30%	System slave select.
LTrdyI	input	(2), 30%	System target ready.
XBRdVld	input	(2), 30%	Crossbar read data valid.
XBRdSize	input	(2), 30%	Split read data size.
SpltRdFull	output	(2), 30%	Read data queue full.
LId	output	(2), 20%	Instruction/data.
LUc	output	(2), 20%	Bus request.
LCoe[9:0]	output	(2), 20%	Command output enable.
LToe	output	(2), 20%	Transaction output enable.
LDoe[7:0]	output	(2), 20%	Data output enable.
LReq	output	(2), 50%	Bus request.
LGnt	input	(2), 30%	Bus grant.
Shared RAM Request/Grant Interface			
EXT_IWREQRAM_R	input	30%	External hardware drives to one to request access to IMEM.
IW_GNTRAM_R	output	30%	Cpu drives to one to grant external IMEM access request.
EXT_DWREQRAM_R	input	30%	External hardware drives to one to request access to DMEM.
DW_GNTRAM_R	output	30%	Cpu drives to one to grant external DMEM access request.
EXT_ICREQRAM_R	input	30%	External hardware drives to one to request access to ICACHE.
IC_GNTRAM_R	output	30%	Cpu drives to one to grant external ICACHE access request.
EXT_DCREQRAM_R	input	30%	External hardware drive to one to request access to DCACHE.

Port Name	I/O	Timing	Description
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 coprocessor.
C<z>rd_gen	output	50%	Cop general register read command.
C<z>rd_con	output	50%	Cop control register read command.
C<z>rd_data[31:0]	input	80%	Cop read data.
C<z>wr_addr[4:0]	output	20%	Cop write address.
C<z>wr_gen	output	20%	Cop general register write command.
C<z>wr_con	output	20%	Cop control write address command.
C<z>wr_data[31:0]	output	30%	Cop write data.
C<z>invld_M	output	60%	Cop invalid instruction flag, one indicates invalid instruction in M stage.
C<z>xcpn_M	output	60%	Cop exception flag, one indicates exception in M stage.
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
Custom Engine Interface			
CEI_CE1HOLD	output	45%	CPU is halting Custom Engine.
CEI_CE1INVLD_M	output	40%	Instruction is not valid, M stage.
CEI_CE1INVLDP_S_R	output	30%	Instruction is not valid, S stage.
CEI_XCPN_M_C1	output	40%	CPU reports exception.
CEI_CE1OP_S_R[11:0]	output	30%	Custom Engine op code.
CEI_INSTM32_S_R_C1_N	output	30%	One indicates 32-bit instruction mode; zero indicates 16-bit instruction mode.

Port Name	I/O	Timing	Description
CEI_CE1AOP_E_R[31:0]	output	35%	A operand.
CEI_CE1BOP_E_R[31:0]	output	35%	B operand.
CE1_RES_E[31:0]	input	45%	Result from Custom Engine.
CE1_SEL_E_R	input	30%	One indicates Custom Engine opcode is present in E stage.
CE1_HALT_E_R[2:0]	input	20%	Custom Engine stalls processor by driving to ones, allows processor to run by driving to zeros. (Copies must be supplied from multiple registers to meet timing requirements.)

Appendix C. LX5280 Pipeline Stalls

This section documents stall conditions that may arise in the LX5280.

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.

Dual-issue interlock: Only one of the potential pair of instructions enters a pipe, the other instruction of the pair waits for the next cycle to enter.

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 50: Instruction Groupings For Stall Definition

Group Name	Instructions in Group
M-I-LoadStore:	LB, LH, LW, LBU, LHU, LWC1, LWC2, LWC3 SB, SH, SW, SWC1, SWC2, SWC3
M-I-Mac	MULT(U),DIV(U),MFHI,MFLO,MTHI,MTLO
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
M16-LoadStore	LB, LH, LWSP, LW, LBU, LHU, LWPC, SB, SH, SWSP, SW, SWRASP
M16-Mac	MULT(U), DIV(U), MFHI, MFLO
M16-Control	JAL(X), JR, JALR, BREAK
M16-UnlinkedBranch	B, BEQZ, BNEZ, BTEQZ, BTNEZ
M16-General	All remaining M16 instructions
RAD-Mac	MTA2, MFA, MFA2, MULTA(U), MULTA2, MULNA2, CMULTA, MADDA(U), MSUBA(U), ADDMA, SUBMA, DIVA(U), MADDA2, MSUBA2, RNDA2
RAD-LoadStore	LT, LTP, LWP, LHP, LBP, LHPU, LBPU, ST, STP, SWP, SHP, SBP
RAD-Control	MTRU, MFRU, MTRK, MFRK, MTLXC0, MFLXC0

Group Name	Instructions in Group
RAD-General	All remaining RAD instructions
EJTAG-Control	DERET, SDBBP, M16SDBBP

C.3. Dual Pipe Issue Rules

These instruction groups must issue to Pipe A:

M-I-LoadStore, M-I-Control, M-I-UnlinkedBranch,
M16-LoadStore, M16-Control, M16-UnlinkedBranch,
RAD-LoadStore, RAD-Control,
EJTAG-Control

These instruction groups must issue to Pipe B:

M-I-Mac, M16-Mac, RAD-Mac

These instruction groups must single issue:

M-I-Control, RAD-Control, EJTAG-Control,
ALL M16 instructions

Instruction doubleword issue rule:

In order for a pair of instructions to dual-issue, they must be found in the same aligned doubleword.

UnlinkedBranch-delay slot rules:

An UnlinkedBranch can dual issue with the preceding instruction, if no other rules are violated. The delay slot instruction of an M-I-UnlinkedBranch single issues in the cycle following the UnlinkedBranch.

Producer-consumer Read-After-Write (RAW) hazard:

A pair of instructions will NOT dual issue if the second instruction uses a register updated by the first instruction. This does not apply to register 0, which never causes an interlock.

Producer-Producer Write-After-Write (WAW) hazard:

A pair of instructions will NOT dual issue if the second instruction updates a register updated by the first instruction. Unless the common target register is also a source register of the second instruction (in which case the RAW interlock applies), no useful program is expected to include such a pair of instructions, since the results of the first update are lost. This does not apply to register 0, which never causes an interlock.

Examples:

```
# Both RAW and WAW apply, causing single issue
00: add s0,s1,s2 ; 04: add s0,s0,s3 ;    2xsingle issue (s0 RAW)

# First instruction does no useful work (visible only in case of exception)
00: add s0,s1,s2 ; 04: add s0,s4,s3 ;   2xsingle issue (s0 WAW)
```

C.4. M16 32-bit Instructions

M16-JAL(X) issues in two consecutive cycles.
M16 Extended instructions issue in two consecutive cycles.

C.5. Non-Sequential Program Flow Issue Stalls

M-I JR, JALR and M16 JR, JALR, AL(X):

Two issue stalls after the delay slot instruction.
(The delay slot instruction always single issues.)

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

NO stall cycles after the delay slot instruction.
(The delay slot instruction always single issues.)

M16 taken branches:

One issue stall after the branch.

M-I not-taken branches:

Two issue stalls after the delay slot instruction.
(The delay slot instruction always single issues.)

M16 not-taken branches:

Three issue stalls after the branch.

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

C.6. Load/Store Rules

Store twinword dual-issue interlock:

The Store Twinword instructions (ST,STP) always single issue. (Because they use 3 of the 4 register file read ports, leaving only one for the other instruction, which usually needs two read ports.)

M16 Load slot issue stall:

There is one unconditional issue stall after any M16 Load instruction. (there is no M16 target register analysis).

Load-use single cycle issue stall:

After a Load instruction to a target register, an instruction which follows the load by one CYCLE and uses the target register of the load will stall issue for one cycle.

Note: The architectural load-delay slot has been eliminated. This issue stall applies even to the

instruction immediately following the load.

This does NOT apply to M16 Loads, since they are always followed by a single cycle issue stall.

Examples:

```
# this executes in 3 cycles:
00: lw s0,0(a0) ; 04: addi a0,4   ;    dual issue
08: add s1,s0   ; 0c: add t1,t2   ;    stall(s0 Load-Use), dual issue


# this executes in 3 cycles:
00: lw s0,0(a0) ; 04: addi a0,4   ;    dual issue
08: add t1,t2   ; 0c: add s1,s0   ;    2xsingle issue (s0 Load-Use)


# this executes in 3 cycles:
00: add t1,t2   ; 04: lw s0,0(a0) ;    dual issue
08: addi a0,4   ; 0c: add s1,s0   ;    2xsingle issue (s0 Load-Use)


# this executes in 3 cycles:
00: lw s0,0(a0) ; 04: lw s2,4(a0) ;    2xPipeA single issue
08: add s1,s0   ; 0c: addi a0,8   ;    dual issue


# this executes in 3 cycles NOTE ELIMINATION OF ARCHITECTURAL DELAY SLOT!
00: add t1,t2   ; 04: lw s0,0(a0) ;    dual issue
08: add s1,s0   ; 0c: addi a0,4   ;    stall(s0 Load-Use), dual issue


# this executes in 2 cycles:
00: add t1,t2   ; 04: lw s0,0(a0) ;    dual issue
08: add s2,s1   ; 0c: addi a0,4   ;    dual issue (s0 not used)
```

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

For Radiax Pointer-Update Load instructions, (LBP,LHP,LTP,LWP,LBPU) this rule does NOT apply to the updated pointer register, which is covered by the RAW and WAW hazard dual-issue interlocks.

Load sub-word stall:

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

Store-Load stall:

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

Note: This stall only applies if the Store instruction hits in the Dcache or has a Byte or Halfword operand.

Examples:

```
# this executes in 3 cycles:
00: sw s0,4(a0) ; 04: addi a0,8   ;    dual issue
08: add s0,s1   ; 0c: lw s2,0(a0) ;    dual issue (and sw-lw stall)
```

```
# this executes in 3 cycles:
00: sw s0,4(a0) ; 04: addi a0,8   ;    dual issue
08: lw s2,0(a0) ; 0c: add s0,s1   ;    dual issue (and sw-lw stall)


# this executes in 4 cycles:
00: sw s0,4(a0) ; 04: lw s2,8(a0) ;    2xPipeA sing issue (and sw-lw stall)
08: addi a0,8   ; 0c: add s0,s1   ;    dual issue


# this executes in 2 cycles:
00: lw s2,0(a0) ; 04: add s0,s1   ;    dual issue
08: sw s0,4(a0) ; 0c: addi a0,8   ;    dual issue  (lw-sw okay)
```

StoreTwin - StoreAny stall:

Any store instruction which follows a Store Twinword instruction (ST,STP) by one CYCLE, always causes a single cycle stall.

Examples:

```
# this executes in 4 cycles:
00: nop         ; 04: st s0,8(a0) ;    2xsingle issue (st single issue)
08: add s0,s1   ; 0c: sw s2,0(a0) ;    dual issue (and st-sw stall)


# this executes in 3 cycles:
00: st s2,0(a0) ; 04: st s0,8(a0) ;    2xsingle issue (and st-st stall)
```

StoreAny - StoreSubword stall:

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. This cycle includes any potential StoreTwin-StoreAny stall.

Examples:

```
# this executes in 3 cycles:
00: sw s0,4(a0) ; 04: addi a0,8   ;    dual issue
08: add s0,s1   ; 0c: sb s2,0(a0) ;    dual issue (and sw-sb stall)


# this executes in 4 cycles:
00: sh s0,4(a0) ; 04: addi a0,8   ;    dual issue (and subword stall)
08: sb s2,0(a0) ; 0c: add s0,s1   ;    dual issue (and sh-sb stall)


# this executes in 3 cycles:
00: st s2,0(a0) ; 04: sb s0,3(a0) ;    2xsingle issue (and st-sb stall)


# this executes in 4 cycles:
00: nop         ; 04: st s0,8(a0) ;    2xsingle issue (st single issue)
08: add s0,s1   ; 0c: sb s2,0(a0) ;    dual issue (and st-sb stall)
```

C.7. Load/Store Ops Stall Matrix

The following table summarizes the stall rules related to Load and Store instructions described above. This

table does NOT include the RAW and WAW dual-issue interlocks. In this table, the "2nd OP" refers to an instruction which issues in the CYCLE after the "1st OP".

Table 51: Load/Store Ops Stall Matrix

2nd op \ 1st op	M16 LW	M16 LB(U) LH(U)	M-I/RAD LB(U) LH(U)	M-I/RAD LBP(U) LHP(U)	SB, SBP SH, SHP	SW SWP	ST STP
non load-store	1U	2U	1S	1U 2S	-	-	-
LT, LTP LW, LWP LB(U) LH(U) LBP(U) LHP(U)	1U	2U	1S	2S	1U	1U	1U
SB, SBP, SH, SHP	1U	2U	1S	1U 2S	1U	1U	1U
SW, SWP ST, STP	1U	2U	1S	2S	-	-	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. Mac Ops Interlock Matrix

The Mac in the 5280 eliminates all programming hazards between Mac instructions by stalling the pipeline as necessary. This is done both to avoid resource conflicts as well as to wait for results of a first instruction that is needed by a second instruction.

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

Table 52: Cycles Required Between Dual MAC Instructions

1st Op / 2nd Op	MULTA(U) MULT(U)	MADDA(U) MSUBA(U) MAD(U) MSUB(U)	CMULTA	DIVA(U) DIV(U)	MADDA2[.S] MSUBA2[.S] ADDMA[.S] SUBMA[.S] MULTA2 MULNA2 RNDA2 MTA2 MTHI MTLO	MFA MFA2 MFHI MFLO
MULTA(U) MADDA(U) MSUBA(U) MAD(U) MSUB(U)	1U	1U	1U	(19T)	-	-
DIVA(U) DIV(U)	(3T)	(4T)	(1T)	19U	-	-
CMULTA MADDA2[.S] MSUBA2[.S] MULTA2 MULNA2 MTA2 MTHI MTLO	3U	4U	1U	(19T)	-	-
ADDMA[.S] SUBMA[.S] RNDA2	LO 2S 2T HI 3S 3T	4U	1S 1T	(19S) (19T)	-	-
MFA MFA2 MFHI MFLO	LO 4S HI 5S	LO 5S HI 6S	3S	19S	2S	-

Nomenclature:

-	=	The two ops can be issued back-to-back.
xU	=	Unconditional delay of the indicated number of cycles.
xS	=	Delay only if (any) 2nd Op source is the same as (any) 1st Op target (producer-consumer dependency).
xT	=	Delay only if (any) 2nd Op target is the same as (any) 1s Op target (preserve write after write order).
( )	=	Items in parenthesis are unlikely to occur in any useful program, which would probably have an intervening MFA.
LO/HI	=	For the 72-bit result of a 32x32 MULT or MADDA, the LO 32-bits (m0l, m1l, etc.) are available one cycle earlier.

Delay of “x” cycles means that if the 1st Op issues in cycle N, then the 2nd Op may issue in cycle N+x+1.

Examples:

```
# this executes in 7 cycles:
00: mult s0,s1  ; 04: addi a0,8   ;    dual issue
08: lw s0,0(a0) ; 0c: lw s1,0(a0) ;    2xPipeA sing issue
10: addi a1,4   ; 14: mflo v0     ;    dual issue (and stall2)
18: sw v0,0(a1) ; 1c: nop         ;    dual issue
```

C.9. MVCz Stall

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

C.10. ZovLoop Rules

There are no special ZovLoop rules but the execution of a ZovLoop must follow all of the other rules as it wraps from the loop end back to the loop start. Unless one of these other rules require it, there are NO stall cycles between loop end and loop start.

Examples:

lps = 00, lpe = 0c unless otherwise noted

```
# executes in 2 cycles per loop BUT GETS THE WRONG ANSWER!
00: add s1,s0   ; 04: addi a0,8   ;    dual issue
08: add t1,t2   ; 0c: lw s0,0(a0) ;    dual issue
00: add s1,s0   ; 04: addi a0,8   ;    dual issue BAD s1 RESULT!
08: add t1,t2   ; 0c: lw s0,0(a0) ;    dual issue


# executes in 3 cycles per loop
00: addi a0,8   ; 04: add s1,s0   ;    dual issue
08: add t1,t2   ; 0c: lw s0,0(a0) ;    dual issue
00: addi a0,8   ; 04: add s1,s0   ;    2x single issue (s0 Load-Use)
08: add t1,t2   ; 0c: lw s0,0(a0) ;    dual issue


# executes in 2 cycles per loop
00: addi a0,8   ; 04: add t1,t2   ;    dual issue
08: add s1,s0   ; 0c: lw s0,0(a0) ;    dual issue
00: addi a0,8   ; 04: add t1,t2   ;    dual issue
08: add s1,s0   ; 0c: lw s0,0(a0) ;    dual issue


# executes in 4 cycles per loop (block copy, one word per loop)
00: addi a0,4   ; 04: lw s0,0(a1) ;    dual issue
08: addi a1,4   ; 0c: sw s0,0(a0) ;    2x single issue (s0 Load-use)
addi a0,4   ; 04: lw s0,0(a1) ;    2x single issue (sw-lw stall)
08: addi a1,4   ; 0c: sw s0,0(a0) ;    2x single issue (s0 Load-use)


# executes in 5 cycles per loop (block copy, two words per loop)
(lps = 00, lpe = 14)
00: lw s0,0(a1) ; 04: lw s1,4(a1) ;    2x PipeA single issue
08: sw s0,0(a0) ; 0c: sw s1,4(a0) ;    2x PipeA single issue
10: addi a0,8   ; 14: addi a1,8   ;    dual issue
00: lw s0,0(a1) ; 04: lw s1,4(a1) ;    2x PipeA single issue
08: sw s0,0(a0) ; 0c: sw s1,4(a0) ;    2x PipeA single issue
10: addi a0,8   ; 14: addi a1,8   ;    dual issue
```

Use pointer-update instructions

```
# executes in 3 cycles per loop (poor)
00: lwp s0,(a0)8; 04: add s1,s2   ;    dual issue
08: lw  s2,4(a0); 0c: add s1,s0   ;    2x single issue (s0 Load-Use)
00: lwp s0,(a0)8; 04: add s1,s2   ;    dual issue (s2 already delayed)
08: lw  s2,4(a0); 0c: add s1,s0   ;    2x single issue (s0 Load-Use)


# executes in 2 cycles per loop (optimum)
00: add s1,s0   ; 04: lwp s0,(a0)8;    dual issue
08: add s1,s2   ; 0c: lw  s2,4(a0);    dual issue
00: add s1,s0   ; 04: lwp s0,(a0)8;    dual issue
08: add s1,s2   ; 0c: lw  s2,4(a0);    dual issue
```

lps = 04, lpe = 10 (poor alignment)

```
# executes in 3 cycles per loop
00:             ; 04: add s1,s0   ;    single issue
08: add t1,t2   ; 0c: lw s0,0(a0) ;    dual issue
10: addi a0,8   ; 14:                  single issue


00:             ; 04: add s1,s0   ;    single issue
08: add t1,t2   ; 0c: lw s0,0(a0) ;    dual issue
10: addi a0,8   ; 14:                  single issue
```

C.11. IMMU Stalls

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.

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 or M16 branch of any kind
M-I J, JAL(X)
EJTAG DERET
M16 EXTEND
M16 JALX first half

C.12. Cache Miss Stalls

Instruction cache miss stall:

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

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

Instruction cache 2-way soft miss stall:

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

If not running in Lock mode, or if the current cache index has no Locked line, set prediction is based onthe LRU bit (predict the non-least recently used set at the current cache index.)

If running in Lock mode, and the current cache index has a Locked line, set prediction is based on the previous Icache access (predict the Locked set if the previous Icache access hit a Locked line and vice versa).

A soft miss always causes a two-cycle stall.

Data cache miss stall:

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

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. Non-Sequential Program Flow Issue Stall Pipeline Diagrams

M-I JR,JALR and M16 JR,JALR,JAL(X):

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

M-I J, JAL(X), 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
                  I  D  S  E  M
```

M16 taken branches:

```
B           I  D  S  E  M  W
notvld         I  .  .  .  .  .
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
                        I  D  S
```

M16 not-taken branches:

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

C.14. Load/Store Stall Pipeline Diagrams

M16 Load slot issue stall:

```
Load       I  D  S  E  M  W
notvld        I  .  .  .  .  .
Load+2           I  D  S  E  M
```

Load-Use single cycle issue stall:

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


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


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


00: lw s0,0(a0)   I  D  S  E  M  W
04: lw s2,4(a0)   I  d  D  S  E  M  W
08: add s1,s0           I  D  S  E  M  W
0c: addi a0,8           I  D  S  E  M  W
```

Load Subword stall:

```
00: lb            I  D  S  E  M  M  W
04: foo1          I  D  S  E  M  M  W
08: foo2             I  D  S  E  E  M  W
0c: foo3             I  D  S  E  E  M  W
10: foo4                I  D  S  S  E  M  W
14: foo5                I  D  S  S  E  M  W

    RHOLD                     X
```

Store-Load stall:

```
00: sw s0,4(a0)   I  D  S  E  M  W
04: addi a0,8     I  D  S  E  M  W
08: add s0,s1        I  D  S  E  M  M  W
0c: lw s2,0(a0)      I  D  S  E  M  M  W
10: foo2                I  D  S  E  E  M  W
14: foo3                I  D  S  E  E  M  W

    RHOLD                        X
```

```
00: sw s0,4(a0)   I  D  S  E  M  W
04: lw s2,8(a0)   I  d  D  S  E  M  M  W
08: addi a0,8        I  D  S  E  E  M  W
0c: add s0,s2        I  D  S  E  E  M  W
10: foo2                I  D  S  S  E  M  W
14: foo3                I  D  S  S  E  M  W

    RHOLD                     X
```

StoreTwin - StoreAny stall:

```
00: nop           I  D  S  E  M  W
04: st s0,8(a0)   I  d  D  S  E  M  W1 W2
08: add s0,s1        I  D  S  E  M  M  W
0c: sw s2,0(a0)      I  D  S  E  M  M  W
10: foo2                I  D  S  E  E  M  W
14: foo3                I  D  S  E  E  M  W

    RHOLD                        X


00: st s2,0(a0)   I  D  S  E  M  W1 W2
04: st s0,8(a0)   I  d  D  S  E  M  M  W1 W2
08: foo2             I  D  S  E  E  M  W
0c: foo3             I  D  S  E  E  M  W
10: foo4                I  D  S  S  E  M  W
14: foo5                I  D  S  S  E  M  W

    RHOLD                     X
```

StoreAny - StoreSubword stall:

```
00: sw s0,4(a0)   I  D  S  E  M  W
04: addi a0,8     I  D  S  E  M  W
08: add s0,s1        I  D  S  E  M  M  W
0c: sb s2,0(a0)      I  D  S  E  M  M  W
10: foo2                I  D  S  E  E  M  W
14: foo3                I  D  S  E  E  M  W

    RHOLD                     X


00: sh s0,4(a0)   I  D  S  E  M  M  W
04: addi a0,8     I  D  S  E  M  M  W
08: sb s2,0(a0)      I  D  S  E  E  M  M  W
0c: add s0,s1        I  D  S  E  E  M  M  W
10: foo2                I  D  S  S  E  E  M  W
14: foo3                I  D  S  S  E  E  M  W

    RHOLD                     X     X


00: st s2,0(a0)   I  D  S  E  M  W1 W2
04: sb s0,3(a0)   I  d  D  S  E  M  M  W
08: foo2             I  D  S  E  E  M  W
0c: foo3             I  D  S  E  E  M  W
10: foo4                I  D  S  S  E  M  W
14: foo5                I  D  S  S  E  M  W

    RHOLD                     X
```

```
00: nop           I  D  S  E  M  W
04: st s0,8(a0)   I  d  D  S  E  M  W1 W2
08: add s0,s1        I  D  S  E  M  M  W
0c: sb s2,0(a0)      I  D  S  E  M  M  W
10: foo2                I  D  S  E  E  M  W
14: foo3                I  D  S  E  E  M  W

    RHOLD                        X
```

C.15. Mac Ops Interlock Pipeline Diagram

```
00: mult s0,s1    I  D  S  E  M  -
04: addi a0,8     I  D  S  E  M  W
08: lw s0,0(a0)      I  D  S  E  M  W
0c: lw s1,0(a0)      I  D  S  E  M  M  M  W
10: addi a1,4           I  D  S  E  E  E  M  W
14: mflo v0             I  D  S  E  E  E  M  W
18: sw v0,0(a1)            I  D  S  S  S  E  M  W
1c: nop                    I  D  S  S  S  E  M  W

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

C.16. MVCz Stall Pipeline Diagrams

```
00: mtc0       I  D  S  E  M  W
    notvld     I  .  .  .  .  .  .
04: foo        I  d  d  D  S  E  M  W
08: foo1          I  D  S  E  M  W
0c: foo2          I  D  S  E  M  W


00: nop        I  D  S  E  M  W
04: mtc0       I  d  D  S  E  M  W
    notvld        I  .  .  .  .  .
08: foo1          I  d  D  S  E  M  W
0c: foo2          I  d  D  S  E  M  W
10: foo3             I  D  S  E  M  W
14: foo4             I  D  S  E  M  W
```

C.17. ZovLoop Pipeline Diagrams

```
00: addi a0,4     I  D  S  E  M  W
04: lw s0,0(a1)   I  D  S  E  M  W
08: addi a1,4        I  D  S  E  M  W
0c: sw s0,0(a0)      I  d  D  S  E  M  W
00: addi a0,4           I  D  S  E  M  W
04: lw s0,0(a1)         I  D  S  E  M  M  W
08: addi a1,4              I  D  S  E  E  M  W
0c: sw s0,0(a0)            I  d  D  S  S  E  M  W

    RHOLD                           X
```

```
00: lw s0,0(a1)   I  D  S  E  M  W
04: lw s1,4(a1)   I  d  D  S  E  M  W
08: sw s0,0(a0)      I  D  S  E  M  W
0c: sw s1,4(a0)      I  d  D  S  E  M  W
10: addi a0,8           I  D  S  E  M  W
14: addi a1,8           I  D  S  E  M  W


00: lwp s0,(a0)8  I  D  S  E  M  W
04: add s1,s2     I  D  S  E  M  W
08: lw s2,4(a0)      I  D  S  E  M  W
0c: add s1,s0        I  d  D  S  E  M  W
00: lwp s0,(a0)8        I  D  S  E  M  W
04: add s1,s2           I  D  S  E  M  W
08: lw s2,4(a0)            I  D  S  E  M  W
0c: add s1,s0              I  d  D  S  E  M  W
```

C.18. Cache Miss Pipeline Diagrams

Icache miss pipeline diagram:

```
00: foo0          I  D  S  E  M  M  M  M  M  M  W
04: foo1          I  D  S  E  M  M  M  M  M  M  W
08: foo2             I  D  S  E  E  E  E  E  E  M  W
0c: foo3             I  D  S  E  E  E  E  E  E  M  W
10: foo4                I ~d  .  .  .  I  D  S  E  M  W
14: foo5                I ~d  .  .  .  I  D  S  E  M  W

    RHOLD                     X  X  X  X  X
```

Icache 2-way soft miss pipeline diagram:

```
00: foo0          I  D  S  E  M  M  M  W
04: foo1          I  D  S  E  M  M  M  W
08: foo2             I  D  S  E  E  E  M  W
0c: foo3             I  D  S  E  E  E  M  W
10: foo4                I ~d  I  D  S  E  M  W
14: foo5                I ~d  I  D  S  E  M  W
18: foo6                      I  D  S  E  M  W
1c: foo7                      I  D  S  E  M  W

    RHOLD                     X  X
```

Dcache miss pipeline diagram:

```
00: foo           I  D  S  E  M  W
04: lw            I  D  S  E  M  .  .  .  .  W
08: foo1             I  D  S  E  M  M  M  M  M  W
0c: foo2             I  D  S  E  M  M  M  M  M  W
08: foo3                I  D  S  E  E  E  E  E  M  W
0c: foo4                I  D  S  E  E  E  E  E  M  W

    RHOLD                     X  X  X  X
```