LatticeMico32 Processor Reference Manual

Lattice Semiconductor Corporation
5555 NE Moore Court
Hillsboro, OR 97124
(503) 268-8000

December 2011

Copyright

Trademarks

Disclaimers

to correct any errors contained herein or to advise any user of this document of any correction if such be made. LSC recommends its customers obtain the latest version of the relevant information to establish, before ordering, that the information being relied upon is current.

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
. . .	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
|	A choice between items in syntax descriptions.

Contents

1

LatticeMico32 Processor and Systems

As systems become more complex, there are a growing number of L_2 and L_3 protocols that continue to burden a local host processor. These tend to incrementally add processing requirements to the local processor, starving other critical functions of processor machine cycles. To alleviate the local host processor's processing requirements, embedded processors are being utilized to support the main processor in a distributed processing architecture. These embedded processors offer localized control, OA&M functionality, and statistics gathering and processing features, thereby saving the host processor many unnecessary clock cycles, which can be used for higher-level functions.

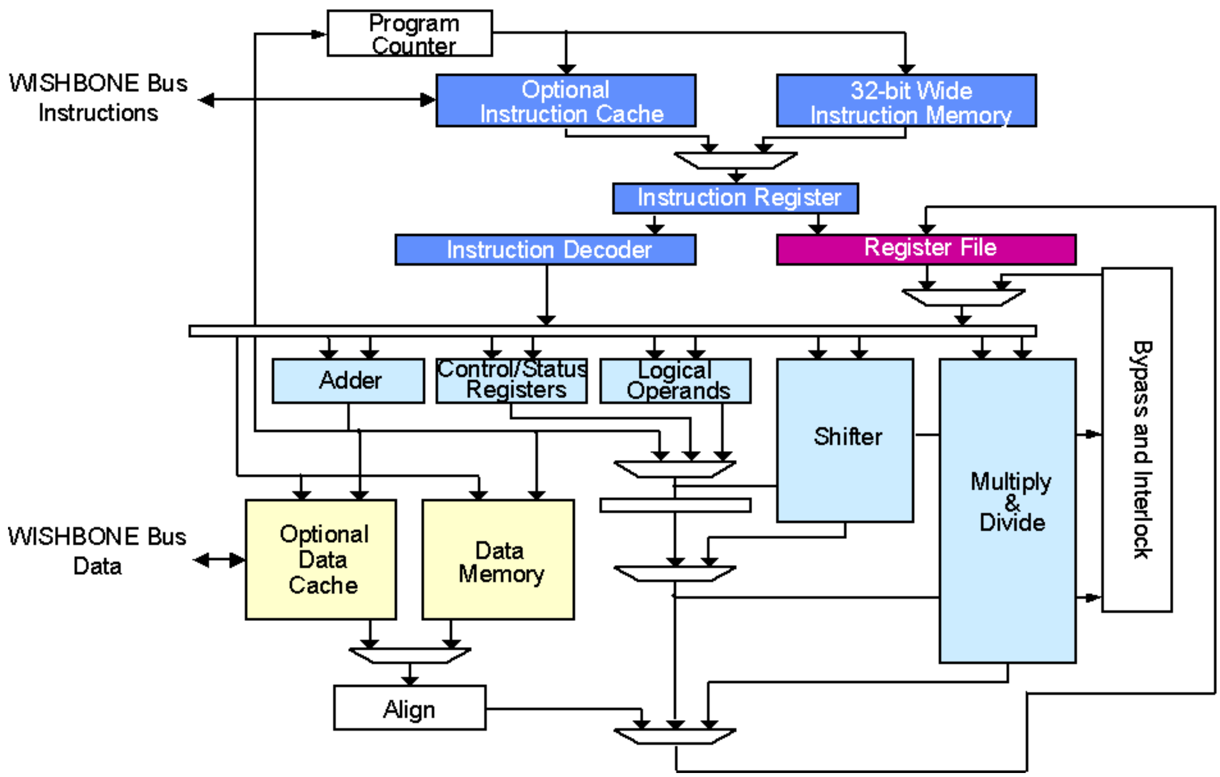
A soft processor provides added flexibility in the implementation of your design. Functionality that can be implemented in software rather than hardware allows much greater freedom in terms of the types of changes that can be made. With software-based processing, it is possible for the hardware logic to remain stable and functional upgrades can be made through software modification. Additionally, it is much quicker and simpler to implement functionality in software than it is to design it in hardware, leading to a reduced time to market.

The LatticeMico32™ is a configurable 32-bit soft processor core for Lattice Field Programmable Gate Array (FPGA) devices. By combining a 32-bit wide instruction set with 32 general-purpose registers, the LatticeMico32 provides the performance and flexibility suitable for a wide variety of markets, including communications, consumer, computer, medical, industrial, and automotive. With separate instruction and data buses, this Harvard architecture processor allows for single-cycle instruction execution as the instruction and data memories can be accessed simultaneously. Additionally, the LatticeMico32 uses a Reduced Instruction Set Computer (RISC) architecture, thereby providing a simpler instruction set and faster performance. As a result, the processor core consumes minimal device resources, while maintaining the performance required for a broad application set. Some of the key features of this 32-bit processor include:

- RISC architecture
- 32-bit data path
- 32-bit instructions
- 32 general-purpose registers
- Up to 32 external interrupts
- Optional instruction cache
- Optional data cache
- Dual WISHBONE memory interfaces (instruction and data)

Figure 1 shows a block diagram of the LatticeMico32 processor core.

Figure 1: LatticeMico32 Block Diagram



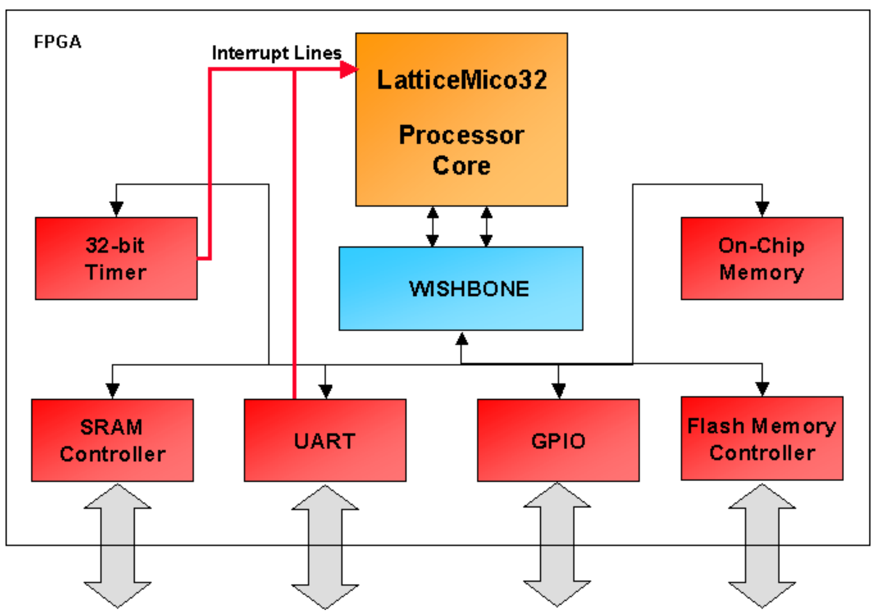
To accelerate the development of processor systems, several optional peripheral components are available with the LatticeMico32 processor. Specifically, these components are connected to the processor through a WISHBONE bus interface, a royalty-free, public-domain specification. By using this open source bus interface, you can incorporate your own

WISHBONE components into your embedded designs. The components include:

- Memory controllers
  - Asynchronous SRAM
  - Double data rate (DDR)
  - On-chip
- Input/output (I/O) ports
  - 32-bit timer
  - Direct memory access (DMA) controller
  - General-purpose I/O (GPIO)
  - I^2C master controller
  - Serial peripheral interface (SPI)
  - Universal asynchronous receiver transmitter (UART)

Figure 2 shows a complete embedded system using the LatticeMico32 processor along with several components.

Figure 2: LatticeMico32 Processor Embedded System



This manual describes the architecture of the LatticeMico32 processor. It includes information on configuration options, pipeline architecture, register architecture, memory architecture, debug architecture, and the instruction set. It is intended to be used as a reference when you design processors for use on supported Lattice Semiconductor field programmable gate arrays (FPGAs).

2

Programmer's Model

This chapter describes the pipeline architecture of the LatticeMico32 processor.

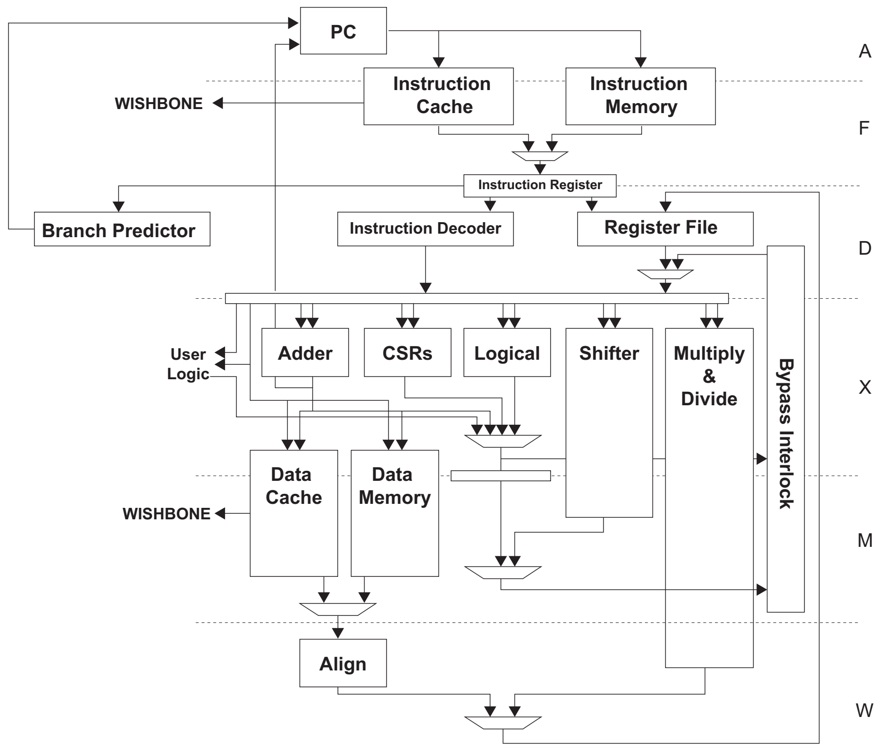
Pipeline Architecture

The LatticeMico32 processor uses a 32-bit, 6-stage pipeline, as shown in Figure 3 on page 6. It is fully bypassed and interlocked. The bypass logic is responsible for forwarding results back through the pipeline, allowing most instructions to be effectively executed in a single cycle. The interlock is responsible for detecting read-after-write hazards and stalling the pipeline until the hazard has been resolved. This avoids the need to insert nop directives between dependent instructions, keeping code size to a minimum, as well as simplifying assembler-level programming.

The six pipeline stages are:

- Address – The address of the instruction to execute is calculated and sent to the instruction cache.
- Fetch – The instruction is read from memory.
- Decode – The instruction is decoded, and operands are either fetched from the register file or bypassed from the pipeline. PC-relative branches are predicted by a static branch predictor.
- Execute – The operation specified by the instruction is performed. For simple instructions such as addition or a logical operation, execution finishes in this stage, and the result is made available for bypassing.
- Memory – For more complicated instructions such as loads, stores, multiplies, or shifts, a second execution stage is required.
- Writeback – Results produced by the instructions are written back to the register file

Figure 3: LatticeMico32 Pipeline



Data Types

The LatticeMico32 processor supports the data types listed in Table 1.

Table 1: Data Types

Type	Range	Bits	Encoding	C Compiler Type
Unsigned byte	$[0, 2^8-1]$	8	Binary	Unsigned character
Signed byte	$[-2^7, 2^7-1]$	8	Two's complement	Character
Unsigned half-word	$[0, 2^{16}-1]$	16	Binary	Unsigned short
Signed half-word	$[-2^{15}, 2^{15}-1]$	16	Two's complement	Short
Unsigned word	$[0, 2^{32}-1]$	32	Binary	Unsigned int/ unsigned long
Signed word	$[-2^{31}, 2^{31}-1]$	32	Two's complement	Int/long

In addition to the above, the extended data types in Table 2 can be emulated through a compiler.

Table 2: Extended Data Types

Data Type	Range	Bits	Encoding	C Compiler Type
Unsigned double-word	[0, 2^{64}-1]	64	Binary	Unsigned long long
Signed double-word	[-2^{63}, 2^{63}-1]	64	Two's complement	Long long
Single-precision real	[1.1754e-38, 3.4028e+38]	32	IEEE 754	Float
Double-precision real	[2.2250e-308, 1.7976e+308]	64	IEEE 754	Double

Register Architecture

This section describes the register architecture of the LatticeMico32 processor.

General-Purpose Registers

The LatticeMico32 features the following 32-bit registers:

- By convention, register 0 (r0) must always hold the value 0, and this is required for correct operation by both the LatticeMico32 assembler and the C compiler. On power-up, the value of 0 in r0 is not hardwired, so you must initialize it to load r0 with the 0 value.
- Registers 1 through 28 are truly general purpose and can be used as the source or destination register for any instruction. After reset, the values in all of these registers are undefined.
- Register 29 (ra) is used by the call instruction to save the return address but is otherwise general purpose.
- Register 30 (ea) is used to save the value of the Program Counter (PC) when an exception occurs, so it should not be used by user-level programs.
- Register 31 (ba) saves the value of the Program Counter (PC) when a breakpoint or watchpoint exception occurs, so it should not be used by user-level programs.

After reset, the values in all of the above 32-bit registers are undefined. To ensure that register 0 contains 0, the first instruction executed after reset should be `xor r0, r0, r0`.

Table 3 lists the general-purpose registers and specifies their use by the C compiler. In this table, the callee is the function called by the caller function.

Table 3: General-Purpose Registers

Register Name	Function	Saver
r0	Holds the value zero	
r1	General-purpose/argument 0/return value 0	Caller
r2	General-purpose/argument 1/return value 1	Caller
r3	General-purpose/argument 2	Caller
r4	General-purpose/argument 3	Caller
r5	General-purpose/argument 4	Caller
r6	General-purpose/argument 5	Caller
r7	General-purpose/argument 6	Caller
r8	General-purpose/argument 7	Caller
r9	General-purpose	Caller
r10	General-purpose	Caller
r11	General-purpose	Callee
r12	General-purpose	Callee
r13	General-purpose	Callee
r14	General-purpose	Callee
r15	General-purpose	Callee
r16	General-purpose	Callee
r17	General-purpose	Callee
r18	General-purpose	Callee
r19	General-purpose	Callee
r20	General-purpose	Callee
r21	General-purpose	Callee
r22	General-purpose	Callee
r23	General-purpose	Callee
r24	General-purpose	Callee
r25	General-purpose	Callee
r26/gp	General-purpose/global pointer	Callee
r27/fp	General-purpose/frame pointer	Callee
r28/sp	Stack pointer	Callee
r29/ra	General-purpose/return address	Caller
r30/ea	Exception address	
r31/ba	Breakpoint address	

Control and Status Registers

Table 4 shows all of the names of the control and status registers (CSR), whether the register can be read from or written to, and the index used when accessing the register. Some of the registers are optional, depending on the configuration of the processor (see "Configuring the LatticeMico32 Processor" on page 33). All signal levels are active high.

Table 4: Control and Status Registers

Name	Access	Index	Optional	Description
PC			No	Program counter
IE	R/W	0x0	Yes	Interrupt enable
IM	R/W	0x1	Yes	Interrupt mask
IP	R	0x2	Yes	Interrupt pending
ICC	W	0x3	Yes	Instruction cache control
DCC	W	0x4	Yes	Data cache control
CC	R	0x5	Yes	Cycle counter
CFG	R	0x6	No	Configuration
EBA	R/W	0x7	No	Exception base address
CFG2	R	0xA	No	Extended configuration

PC – Program Counter

The PC CSR is a 32-bit register that contains the address of the instruction currently being executed. Because all instructions are four bytes wide, the two least significant bits of the PC are always zero. After reset, the value of the PC CSR is h00000000.

Figure 4: Format of the PC CSR

31 … 2	1	0
PC	0	0

IE – Interrupt Enable

The IE CSR contains a single-bit flag, IE, that determines whether interrupts are enabled. This flag has priority over the IM CSR. In addition, there are two bits, BIE and EIE, that are used to save the value of the IE field when either a breakpoint or other exception occurs. Each interrupt is associated with a mask bit (IE bit) indexed with each interrupt. After reset, the value of the IE CSR is h00000000.

Figure 5: Format of the IE CSR

31 … 3	2	1	0
-	BIE	EIE	IE

Table 5: Fields of the IE CSR

Field	Values	Description
IE	0 – Interrupts disabled 1 – Interrupts enabled	Determines whether interrupts are enabled.
EIE	0 – Interrupts disabled 1 – Interrupts enabled	Holds a copy of the IE field when an exception occurs.
BIE	0 – Interrupts disabled 1 – Interrupts enabled	Holds a copy of the IE field when a breakpoint occurs.

IM – Interrupt Mask

The IM CSR contains an enable bit for each of the 32 interrupts. Bit 0 corresponds to interrupt 0. In order for an interrupt to be raised, both an enable bit in this register and the IE flag in the IE CSR must be set to 1. After reset, the value of the IM CSR is h00000000.

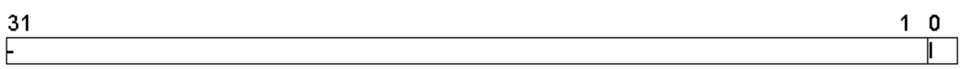
IP – Interrupt Pending

The IP CSR contains a pending bit for each of the 32 interrupts. A pending bit is set when the corresponding interrupt request line is asserted low. Bit 0 corresponds to interrupt 0. Bits in the IP CSR can be cleared by writing a 1 with the wcsr instruction. Writing a 0 has no effect. After reset, the value of the IP CSR is h00000000.

ICC – Instruction Cache Control

The ICC CSR provides a control bit that, when written with any value, causes the contents of the entire instruction cache to be invalidated.

Figure 6: Format of the ICC CSR



Field	Values	Description
I	Any – Invalidate instruction cache	When written, the contents of the instruction cache are invalidated.

DCC – Data Cache Control

The DCC CSR provides a control bit that, when written with any value, causes the contents of the entire data cache to be invalidated.

Figure 7: Format of the DCC CSR

31 – 1	0
-	I

Table 6: Fields of the DCC CSR

Field	Values	Description
I	Any – Invalidate data cache	When written, the contents of the data cache are invalidated.

CC – Cycle Counter

The CC CSR is an optional 32-bit register that is incremented on each clock cycle. It can be used to profile ghost code sequences.

CFG – Configuration

The CFG CSR details the configuration of a particular instance of a LatticeMico32 processor.

Figure 8: Format of the CFG CSR

31 – 26	24 – 22	21 – 18	17 – 12	11	10	9	8	7	6	5	4	3	2	1	0
REV	WP	BP	INT	J	R	H	G	IC	DC	CC	X	U	S	D	M

Table 7: Fields of the CFG CSR

Field	Values	Description
M	0 – Multiply is not implemented 1 – Multiply is implemented	Indicates whether a hardware multiplier is implemented.
D	0 – Divide is not implemented 1 – Divide is implemented	Indicates whether a hardware divider is implemented.
S	0 – Barrel shift is not implemented 1 – Barrel shift is implemented	Indicates whether a hardware barrel-shifter is implemented.
U		Reserved.
X	0 – Sign extend is not implemented 1 – Sign extend is implemented	Indicates whether the sign-extension instructions are implemented.
CC	0 – Cycle counter is not implemented 1 – Cycle counter is implemented	Indicates whether the CC CSR is implemented.
IC	0 – Instruction cache is not implemented 1 – Instruction cache is implemented	Indicates whether an instruction cache is implemented.

Table 7: Fields of the CFG CSR (Continued)

Field	Values	Description
DC	0 – Data cache is not implemented 1 – Data cache is implemented	Indicates whether a data cache is implemented.
G	0 – Debug is not implemented 1 – Data cache is implemented	Indicates whether software-based debug support is implemented.
H	0 – H/W debug is not implemented 1 – H/W debug is implemented	Indicates whether hardware-based debug support is implemented.
R	0 – ROM debug is not implemented 1 – ROM debug is implemented	Indicates whether support for debugging ROM-based programs is implemented.
J	0 – JTAG UART is not implemented 1 – JTAG UART is implemented	Indicates whether a JTAG UART is implemented.
INT	0 – 32	Indicates the number of external interrupts.
BP	0 – 4	Indicates the number of breakpoint CSRs.
WP	0 – 4	Indicates the number of watchpoint CSRs.
REV	0 – 63	Processor revision number. This is set automatically. You cannot reset this field.

CFG2 – Extended Configuration

The CFG2 CSR is used in conjunction with CFG CSR to provide details on the configuration of a particular instance of the LatticeMico32 processor.

Figure 9: Format of CFG2 CSR

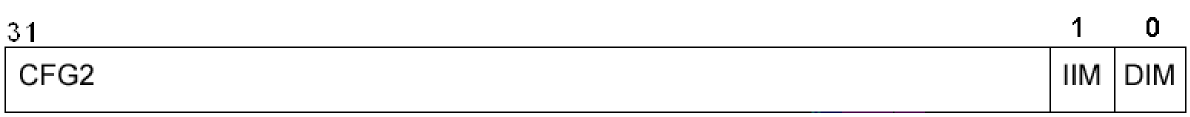


Table 8:

Field	Values	Description
DIM	0 – Data inline memory is not implemented. 1 – Data inline memory is implemented.	Indicates whether data inline memory is implemented.
IIM	0 – Instruction inline memory is not implemented. 1 – Instruction inline memory is implemented.	Indicates whether instruction inline memory is implemented.

EBA – Exception Base Address

The EBA CSR specifies the base address of the exception handlers. After reset, the value of EBA is set to EBA_RESET. If you write a value to the register where the lower byte is not zero, it will read back all zeros. There is no need for you to mask zeros to avoid issues.

Figure 10: Format of EBA CSR

31 ... 8	7	6	5	4	3	2	1	0
EBA	0	0	0	0	0	0	0	0

Memory Architecture

This section describes the memory architecture of the LatticeMico32 processor.

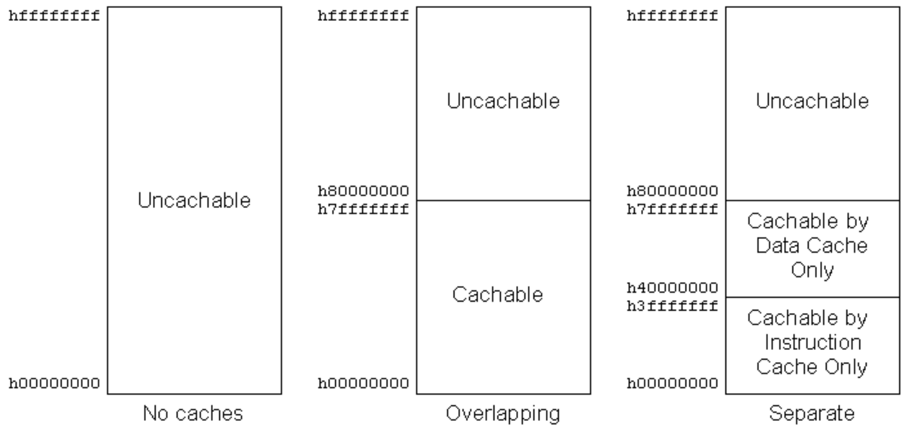
Address Space

The LatticeMico32 processor has a flat 32-bit, byte-addressable address space. By default, this address space is uncachable. The designer can configure the entire address space, or just a portion of it, to be cachable. The designer can also designate the entire uncachable address space, or just a portion of it, to be processor inline memory space. For LatticeMico32 processors with caches, the portion of the address space that is cacheable can be configured separately for both the instruction and data cache. This allows for the size of the cache tag RAMs to be optimized to be as small as is required (the fewer the number of cacheable addresses, the smaller the tag RAMs will be).

If an instruction cache is used, attempts to fetch instructions from outside of the range of cacheable addresses result in undefined behavior, so only one cached region is supported. Portions of the memory image are not cached, so if a miss occurs, it will not be fetched.

Figure 11 illustrates some possible configurations. Typically, the parts of the address space that are cacheable are used for storing code or program data, with I/O components being mapped into uncacheable addresses.

Figure 11: Cacheable Addresses



Endianness

The LatticeMico32 processor is big-endian, which means that multi-byte objects, such as half-words and words, are stored with the most significant byte at the lowest address.

Address Alignment

All memory accesses must be aligned to the size of the access, as shown in Table 9. No check is performed for unaligned access. All unaligned accesses result in undefined behavior.

Table 9: Memory Access Alignment Requirements

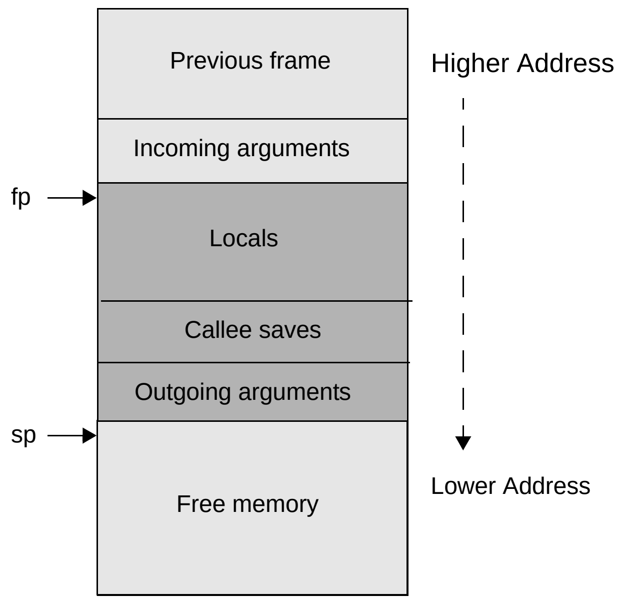
Access Size	Address Requirements
Byte	None
Half-word	Address must be half-word aligned (bit 0 must be 0)
Word	Address must be word aligned (bits 1 and 0 must be 0)

Stack Layout

Figure 12 shows the conventional layout of a stack frame. The stack grows toward lower memory as data is pushed onto it. The stack pointer (sp) points to the first unused location, and the frame pointer (fp) points at the first location used in the active frame. In many cases, a compiler may be able to eliminate the frame pointer, because data can often be accessed by using a negative displacement from the stack pointer, freeing up the frame pointer for use as a general-purpose register.

As illustrated in Table 3 on page 8, the first eight function arguments are passed in registers. Any remaining arguments are passed on the stack, as illustrated in Figure 12.

Figure 12: Stack Layout



Caches

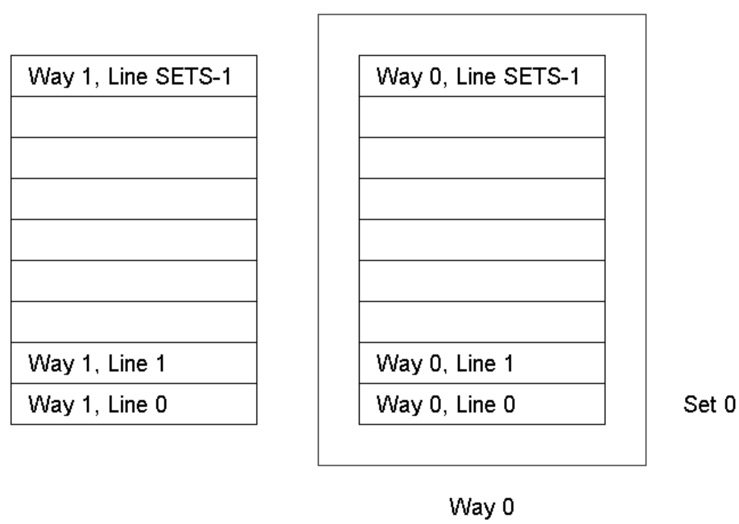
A cache is a fast memory (single-cycle access) that stores a copy of a limited subset of the data held in main memory, which may take the CPU several cycles to access. A cache helps improve overall performance by exploiting the fact that the same data is typically accessed several times in a short interval. By storing a local copy of the data in the processor's cache, the multiple cycles required to access the data can be reduced to just a single cycle for all subsequent accesses once the data is loaded into the cache.

Cache Architecture

When a cache accesses a data item, it is also likely to access data at adjacent addresses (such as with arrays or structures) by loading data into the cache in lines. A line can consist of 4, 8, or 16 adjacent bytes, and is specified by the BYTES_PER_LINE option.

A one-way associative (direct-mapped) cache consists of an array of cache lines known as a "way." To allow the cache to operate at a high frequency, data from main memory can only be stored in a specific cache line. A two-way associative cache consists of a two-dimensional array of cache lines. It requires slightly more logic to implement but allows data from main memory to be stored in one of two places in the cache. It helps performance by reducing cache conflicts that occur when a program is accessing multiple data items that would map to the same cache line in a one-way associative cache. The number of lines in each way is specified by the ICACHE_SETS and DCACHE_SETS options. The ways are assigned in a round-robin fashion. Each time a cache miss occurs the way number is switched.

Figure 13: Cache Organization



The LatticeMico32 caches are write-through, which means that whenever a store instruction writes to an address that is cached, the data is written to both the cache and main memory. A read-miss allocation policy means that a cache line is only fetched from memory for a load instruction. If a cache miss occurs for a store instruction, the data is written directly to memory without the cache being updated.

The LatticeMico32 processor supports a range of cache configurations, as detailed in Table 10.

Table 10: Cache Configurations

Attribute	Values
Size	0 kB, 1 kB, 2 kB, 4 kB, 8 kB, 16 kB, 32 kB
Sets	128, 256, 512, 1024
Associativity	1, 2
Bytes-per-line	4, 8, 16
Write policy	Write-through
Update policy	Read miss only

The LatticeMico32 caches are initialized automatically by embedded logic, so they do not require a program to initialize or enable them.

Invalidating the Caches

The contents of the instruction cache can be invalidated by writing to the ICC CSR. It is recommended that you follow the write to the ICC CSR with four nops, as follows:

```
wcsr ICC, r0
nop
nop
nop
nop
```

The contents of the data cache can similarly be invalidated by writing to the DCC CSR as follows:

```
wcsr DCC, r0
```

It is recommended that you avoid placing a load or store instruction immediately before or after the wcsr instruction.

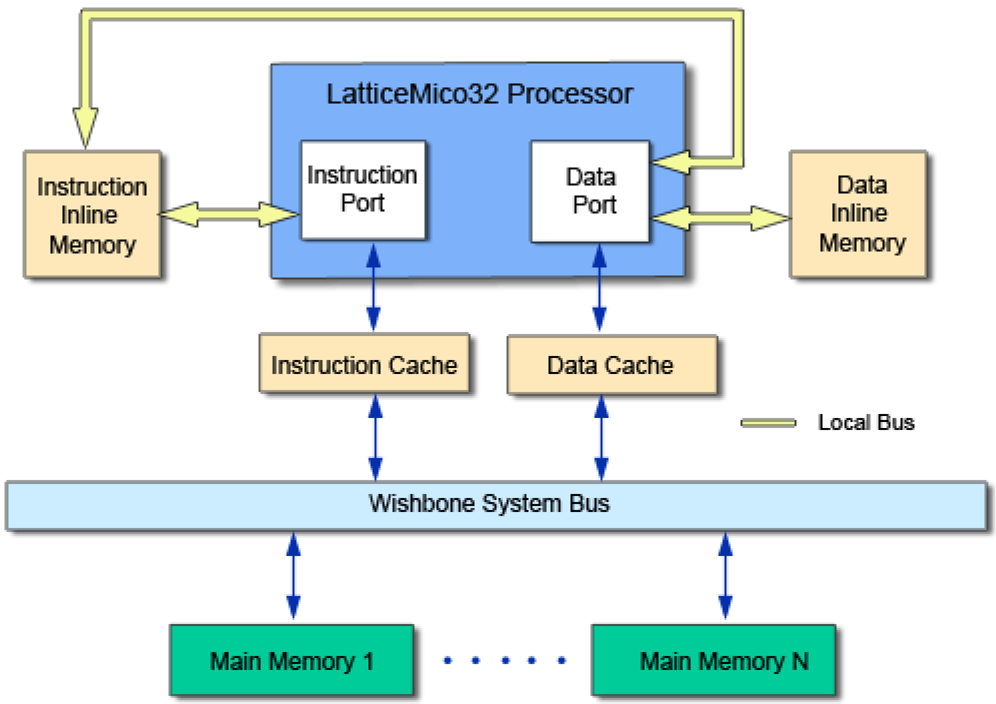
The LatticeMico32 caches are not kept consistent with respect to each other. This means that if a store instruction writes to an area of memory that is currently cached by the instruction cache, the instruction cache will not be automatically updated to reflect the store. It is your responsibility to invalidate the instruction cache after the write has taken place, if necessary.

Similarly, the caches do not snoop bus activity to monitor for writes by peripherals (by DMA for example) to addresses that are cached. It is again your responsibility to ensure that the cache is invalidated before reading memory that may have been written by a peripheral.

Inline Memories

The LatticeMico32 processor enables you to optionally connect to on-chip memory, through instruction and data ports, by using a local bus rather than the Wishbone interface. Memory connected to the CPU in such a manner is referred to as inline memory. Figure 14 shows a functional block diagram of the LatticeMico32 processor with inline memories. The addresses occupied by inline memories are not cachable.

Figure 14: LatticeMico32 Inline Memories



There are two types of inline memories:

- Instruction Inline Memory – This memory component is connected to the Instruction Port of the LatticeMico32 CPU and is used to hold only program memory of any software application.
- Data Inline Memory – This memory component is connected to the Data Port of the LatticeMico32 CPU and is used to hold read-only or read/write data of any software application.

Note

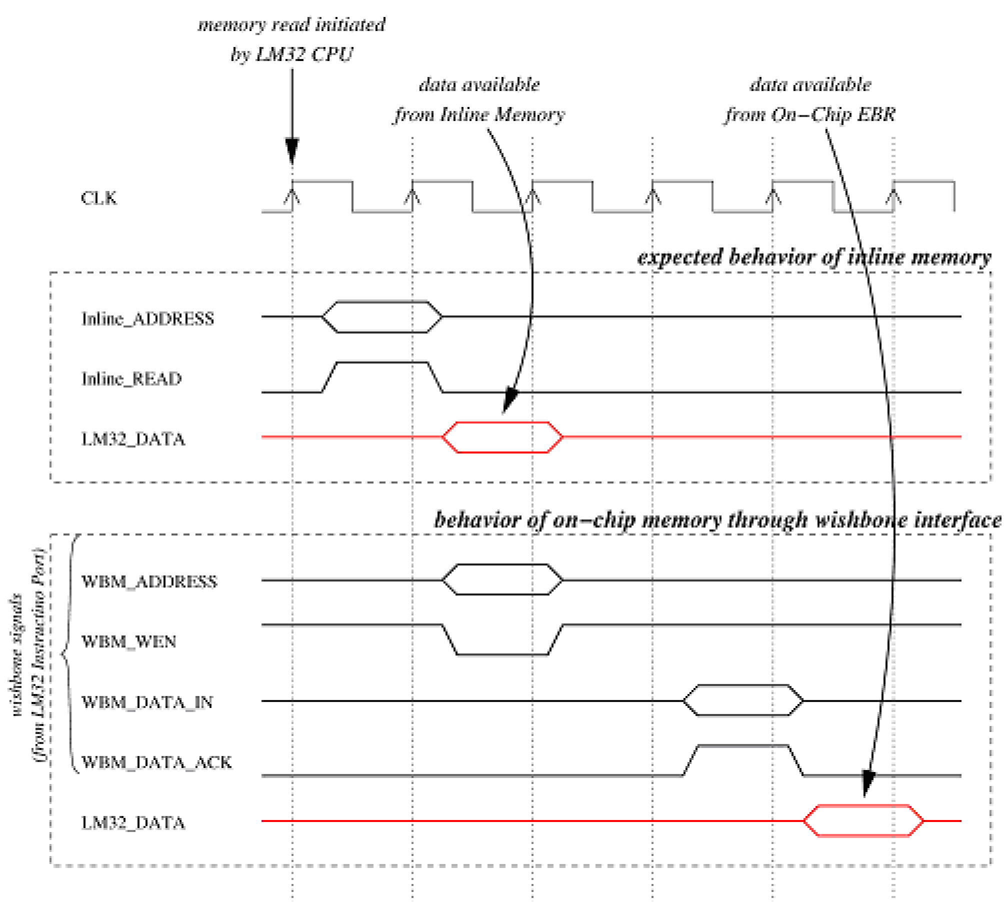
The Instruction Inline Memory is also connected to the Data Port of the LatticeMico32 CPU in order to facilitate loading of the memory image of the software application through the command line lm32-elf-gdb or through the C/C++ SPE Debugger.

While it is possible to create a LatticeMico32 platform that contains inline memories as the sole memory components, inline memories can co-exist in a platform with other Wishbone-based memory components. Inline memories act as types of main memories, but with the difference that the contents of these memories are not cached.

Performance Advantage Over Wishbone-based Memory without Caches

The direct connection between CPU and EBR-based inline memory has the advantage of providing a single-cycle read/write access to the CPU. Figure 15 shows cycle-level analysis of potential performance benefits of inline memory when compared to on-chip memory (EBR) that is connected to the CPU through the Wishbone interface.

Figure 15: Cycle-level Analysis



This diagram compares the number of cycles it takes to service read access from the LatticeMico32 CPU by the inline memory versus the Wishbone-based on-chip EBR. The read access initiated to inline memory will be completed in the next cycle, whereas a read access initiated to EBR will take four cycles. A similar behavior can be seen for writes initiated by the

LatticeMico32 CPU. This shows that deploying program code or data to inline memory can provide at least a 3x speedup over Wishbone-based memories.

Performance Advantage Over Wishbone-based Memory with Caches

It is common to configure the LatticeMico32 CPU with Instruction and Data caches to reduce the performance impact of accessing Wishbone-based memories, since they theoretically provide a single-cycle access. In practice, however, you will encounter situations in which a single-cycle cache access is not possible. In these situations, inline memory affords a performance advantage. Such situations include the following scenarios:

- Any cache access (read or write) that results in a miss will initiate an access to memory components on the Wishbone Interface. As a result, the cache access will not take multiple cycles to complete.
- The data cache in LatticeMico32 is write-through, meaning that any write to the data cache from LatticeMico32 will immediately result in access to memory components on the Wishbone interface. This means that all data cache writes are multicycle accesses.

Exceptions

Exceptions are events either inside or outside of the processor that cause a change in the normal flow of program execution. The LatticeMico32 processor can raise eight types of exceptions, as shown in Table 11. The exceptions are listed in a decreasing order of priority, so if multiple exceptions occur simultaneously, the exception with the highest priority is raised.

Table 11: Exceptions

Exception	ID	Condition
Reset	0	Raised when the processor's reset pin is asserted.
Breakpoint	1	Raised when either a break instruction is executed or when a hardware breakpoint is triggered.
InstructionBusError	2	Raised when an instruction fetch fails, typically due to the requested address being invalid.
Watchpoint	3	Raised when a hardware watchpoint is triggered.
DataBusError	4	Raised when a data access fails, typically because either the requested address is invalid or the type of access is not allowed.
DivideByZero	5	Raised when an attempt is made to divide by zero.
Interrupt	6	Raised when one of the processor's interrupt pins is asserted, providing that the corresponding field in the interrupt mask (IM) CSR is set and the global interrupt enable flag, IE.IE, is set.
SystemCall	7	Raised when an scall instruction is executed.

Exception Processing

Exceptions occur in the execute pipeline stage. It is possible to have two exceptions occur simultaneously. In this situation the exception with the highest priority is handled. The sequence of operations performed by the processor after an exception depends on the type of the highest priority exception that has occured. Before the exception is handled, all instructions in the Memory and Writeback pipeline stages are allowed to complete. Also, all instructions in the Execute, Address, Fetch, and Decode pipeline stages are squashed to ensure that they do not modify the processor state.

Exceptions are categorized in to two broad categories:

- Non-Debug Exceptions
- Debug Exceptions.

Non-Debug Exceptions

The Reset, Instruction Bus Error, Data Bus Error, Divide-By-Zero, Interrupt and System Call exceptions are classified as non-debug exceptions. The following sequence of events occur in one atomic operation:

```
ea = PC
IE.EIE = IE.IE
IE.IE = 0
PC = (DC.RE ? DEBA : EBA) + (ID * 32)
```

Debug Exceptions

The Breakpoint and Watchpoint exceptions are classified as debug exceptions. The following sequence of events occur in one atomic operation:

```
ba = PC
IE.BIE = IE.IE
IE.IE = 0
PC = DEBA + (ID * 32)
```

Exception Handler Code

As seen above, the processor branches to an address that is an offset from either the EBA CSR or the DEBA CSR in order to handle the exception. The offset is calculated by multiplying the exception ID by 32. Exception IDs are shown in Table 11 on page 20. Since all LatticeMico32 instructions are four bytes long, this means each exception handler can be eight instructions long. If further instructions are required, the handler can call a subroutine.

Whether the EBA or DEBA is used as the base address depends upon the type of the exception that occurred, whether DC.RE is set, and whether dynamic mapping of EBA to DEBA is enabled via the 'at_debug' input pin to the processor. Having two different base addresses for the exception table allows a debug monitor to exist in a different memory from the main program code. For example, the debug monitor may exist in an on-chip ROM, whereas the main program code may be in a DDR or SRAM. The DC.RE flag and at_debug pin allow either interrupts to run at full speed when debugging or for the debugger to take complete control and handle all exceptions. When an exception occurs, the only state that is automatically saved by the CPU is the PC, which is saved in either ea or ba, and the interrupt enable flag, IE.IE, which is saved in either IE.EIE or IE.BIE. It is the responsibility of the exception handler to save and restore any other registers that it uses, if it returns to the previously executing code.

The piece of code in Figure 16 shows how the exception handlers can be implemented. The nops are required to ensure that the next exception handler is aligned at the correct address. To ensure that this code is at the correct address, it is common practice to place it in its own section. Place the following assembler directive at the start of the code:

```
.section    .boot, "ax", @progbits
```

Figure 16: Exception Handler Example

```
/* Exception handlers */

_reset_handler:
        xor     r0, r0, r0
        bi      _crt0
        nop
        nop
        nop
        nop
        nop
        nop

_breakpoint_handler:
        sw      (sp+0), ra
        calli   save_all
        mvi     r1, SIGTRAP
        calli   raise
        bi      restore_all_and_bret
        nop
        nop
        nop

_instruction_bus_error_handler:
        sw      (sp+0), ra
        calli   save_all
        mvi     r1, SIGSEGV
        calli   raise
        bi      restore_all_and_eret
        nop
        nop
        nop

_watchpoint_handler:
        sw      (sp+0), ra
        calli   save_all
        mvi     r1, SIGTRAP
        calli   raise
        bi      restore_all_and_bret
        nop
        nop
        nop
```

Figure 16: Exception Handler Example (Continued)

```
_data_bus_error_handler:
        sw      (sp+0), ra
        calli   save_all
        mvi     r1, SIGSEGV
        calli   raise
        bi      restore_all_and_eret
        nop
        nop
        nop

_divide_by_zero_handler:
        sw      (sp+0), ra
        calli   save_all
        mvi     r1, SIGFPE
        calli   raise
        bi      restore_all_and_eret
        nop
        nop
        nop

_interrupt_handler:
        sw      (sp+0), ra
        calli   save_all
        mvi     r1, SIGINT
        calli   raise
        bi      restore_all_and_eret
        nop
        nop
        nop

_system_call_handler:
        sw      (sp+0), ra
        calli   save_all
        mv      r1, sp
        calli   handle_scall
        bi      restore_all_and_eret
        nop
        nop
        nop
```

Figure 16: Exception Handler Example (Continued)

```
_save_all:
        addi    sp, sp, -56
        /* Save all caller save registers onto the stack */
        sw      (sp+4), r1
        sw      (sp+8), r2
        sw      (sp+12), r3
        sw      (sp+16), r4
        sw      (sp+20), r5
        sw      (sp+24), r6
        sw      (sp+28), r7
        sw      (sp+32), r8
        sw      (sp+36), r9
        sw      (sp+40), r10
        sw      (sp+48), ea
        sw      (sp+52), ba
        /* ra needs to be moved from initial stack location */
        lw      r1, (sp+56)
        sw      (sp+44), r1
        ret

/* Restore all registers and return from exception */
_restore_all_and_eret:
        lw      r1, (sp+4)
        lw      r2, (sp+8)
        lw      r3, (sp+12)
        lw      r4, (sp+16)
        lw      r5, (sp+20)
        lw      r6, (sp+24)
        lw      r7, (sp+28)
        lw      r8, (sp+32)
        lw      r9, (sp+36)
        lw      r10, (sp+40)
        lw      ra, (sp+44)
        lw      ea, (sp+48)
        lw      ba, (sp+52)
        addi    sp, sp, 56
        eret

/* Restore all registers and return from breakpoint */
_restore_all_and_bret:
        lw      r1, (sp+4)
        lw      r2, (sp+8)
        lw      r3, (sp+12)
        lw      r4, (sp+16)
        lw      r5, (sp+20)
        lw      r6, (sp+24)
        lw      r7, (sp+28)
        lw      r8, (sp+32)
        lw      r9, (sp+36)
        lw      r10, (sp+40)
        lw      ra, (sp+44)
        lw      ea, (sp+48)
        lw      ba, (sp+52)
        addi    sp, sp, 56
        bret
```

Then in the linker script, place the code at the reset value of EBA or DEBA, as shown in Figure 17.

Figure 17: Placing Exception Handler in Memory

```
MEMORY
{
  ram : ORIGIN = 0x00000000, LENGTH = 0x00100000
}

SECTIONS
{
  .boot : { *(.boot) } > ram
}
```

Nested Exceptions

Because different registers are used to save a state when a debug-related exception occurs (ba and IE.BIE instead of ea and IE.EIE), limited nesting of exceptions is possible, allowing the interrupt handler code to be debugged. Any further nesting of exceptions requires software support.

To enable nested exceptions, an exception handler must save all the state that is modified when an exception occurs, including the ea and ba registers, as well as the IE CSR. These registers can simply be saved on the stack. When returning from the exception handler, these registers must, obviously, be restored from the values saved on the stack.

Nested Prioritized Interrupts

The LatticeMico32 microprocessor supports up to 32 maskable, active-low, level-sensitive interrupts. Each interrupt line has a corresponding mask bit in the IM CSR. The mask enable is active high. A global interrupt enable flag is implemented in the IE CSR. Software can query the status of the interrupts and acknowledge them through the IP CSR.

To support nested prioritized interrupts, an exception handler should save the registers just outlined, then save the IM CSR, and then mask all lower-priority interrupts. IE.IE can then be set to re-enable interrupts. When the interrupt handler has finished, IE.IE should be cleared before all the saved registers, including IM, are restored.

Remapping the Exception Table

In order to increase performance, the exception table can be remapped at run time by writing a new value to EBA. It would be used in a system in which the power-up value of EBA points to a slow, non-volatile memory, such as a FLASH memory, but the code is executed from a faster, non-volatile RAM, such as DDR or SRAM.

Reset Summary

During reset, the following occurs:

- All CSRs are set to their reset values as listed in "Control and Status Registers" on page 9.
- Interrupts are disabled.
- All hardware breakpoints and watchpoints are disabled.
- If implemented, the contents of the caches are invalidated.
- A reset exception is raised, which causes the PC to be set to the value in the EBA CSR, where program execution starts. The PC can be optionally set to the value in the DEBA CSR by enabling dynamic mapping of exception handlers to Debugger (i.e., mapping EBA to DEBA) and asserting the at_debug pin.

The register file is not reset, so it is the responsibility of the reset exception handler to set register 0 to 0. This should be achieved by executing the following sequence: `xor r0, r0, r0`.

Using Breakpoints

The LatticeMico32 architecture supports both software and hardware breakpoints. Software breakpoints should be used for setting breakpoints in code that resides in volatile memory, such as DDR or SRAM, while hardware breakpoints should be used for setting breakpoints in code that resides in non-volatile memory, such as FLASH or ROM.

A software breakpoint is simply a break instruction. In order to set a breakpoint, it is simply a case of replacing the instruction at the desired address with the break instruction. When the break instruction is executed, a breakpoint exception is raised, and the ba register contains the address of the break instruction that was executed. It is then up to the exception handler to either restore the instruction that was overwritten and continue execution, or to take some other action, depending upon why the breakpoint was set.

It is typically either not possible or very slow to write a break instruction to non-volatile RAM. For processors with breakpoints greater than 0, it is possible to set a hardware breakpoint by writing the address of the instruction on which the breakpoint should be set to one of the BP*n* CSRs. The processor then constantly compares the values in these BP*n* CSRs with the address of the instruction being executed. If a match occurs, and the breakpoint is enabled (by the LSB being set to 1), a breakpoint exception will be raised. As with software breakpoints, the address of the instruction that caused the breakpoint is saved in the ba register. If the breakpoint exception handler wishes to resume program execution, it must clear the enable bit in the relevant BP CSR; otherwise, the breakpoint exception is raised as soon as execution resumes.

Using Watchpoints

The LatticeMico32 architecture supports hardware watchpoints. Watchpoints are a mechanism by which a program can watch out for specific memory accesses. For example, a program can set up a watchpoint that will cause a watchpoint exception to be raised every time the address 0 is accessed (something that is useful for tracking down null pointer errors in C programs).

To set up a watchpoint, the memory address that is being watched must be written to one of the WP*n* CSRs. The watchpoint then needs to be enabled by writing the corresponding C field in the DC CSR. This field takes one of the four values that indicate the following:

- The watchpoint is disabled.
- The watchpoint exception is only raised on read accesses.
- The watchpoint exception is only raised on write accesses.
- The watchpoint exception is raised on either read or write accesses.

Debug Architecture

This section describes the debug architecture of the LatticeMico32 processor.

The LatticeMico32 debug architecture provides:

- Software breakpoints
- Hardware breakpoints
- Hardware watchpoints
- Single-step capability
- Ability to remap exception handlers when debugging is enabled
- Hardware support for debugging interrupt handlers

Table 12 shows the debug control and status registers.

Table 12: Debug Control and Status Registers

Name	Access	Index	Description
DC	W	0x8	Debug control
DEBA	R/W	0x9	Debug exception base address
JTX	R/W	0xe	JTAG UART transmit
JRX	R/W	0xf	JTAG UART receive
BP0	W	0x10	Breakpoint address 0
BP1	W	0x11	Breakpoint address 1
BP2	W	0x12	Breakpoint address 2
BP3	W	0x13	Breakpoint address 3

Table 12: Debug Control and Status Registers (Continued)

Name	Access	Index	Description
WP0	W	0x18	Watchpoint address 0
WP1	W	0x19	Watchpoint address 1
WP2	W	0x1a	Watchpoint address 2
WP3	W	0x1b	Watchpoint address 3

DC – Debug Control

The DC CSR contains flags that control debugging facilities. After reset, the value of the DC CSR is h00000000. This CSR is only implemented if DEBUG_ENABLED equals TRUE.

Figure 18: Format of the DC CSR

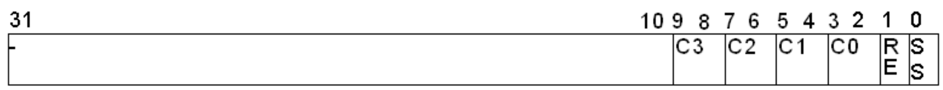


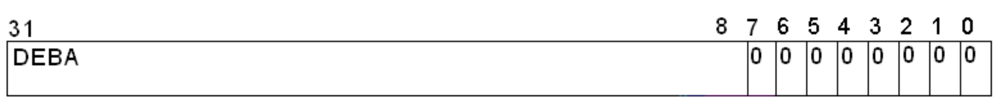
Table 13: Fields of the DC CSR

Field	Value	Description
SS	0 – Single step disabled 1 – Single step enabled	Determines whether single-stepping is enabled
RE	0 – Remap only debug exceptions 1 – Remap all exceptions	Determines whether all exceptions are remapped to the base address specified by DEBA or just debug exceptions
C*n*	b00 – Watchpoint *n* disabled b01 – Break on read b10 – Break on write b11 – Break on read or write	Enable for corresponding Wp*n* CSR

DEBA – Debug Exception Base Address

The DEBA CSR specifies the base address of the debug exception handlers. After reset, the value of the DEBA CSR is set to DEBA_RESET. This CSR is only implemented if DEBUG_ENABLED equals TRUE.

Figure 19: Format of the DEBA CSR



JTX – JTAG UART Transmit Register

The JTX CSR can be used for transmitting data through a JTAG interface. This CSR is only implemented if JTAG_UART_ENABLED equals TRUE.

Figure 20: Format of the JTX CSR

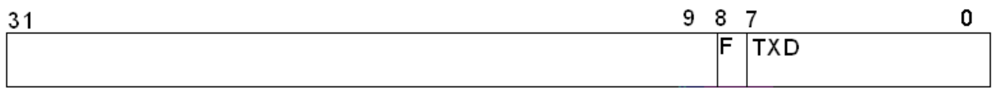


Table 14: Fields of the JTX CSR

Field	Values	Description
TXD		Transmits data
F	0 – Empty 1 – Full	Indicates whether the transmit data register is full

JRX – JTAG UART Receive Register

The JRX CSR can be used for receiving data through a JTAG interface. This CSR is only implemented if JTAG_UART_ENABLED equals TRUE.

Figure 21: Format of the JRX CSR

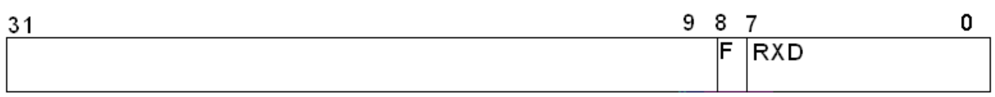


Table 15: Fields of the JTX CSR

Field	Values	Description
RXD		Receives data.
F	0 – Empty 1 – Full	Indicates whether the receive data register is full.

BP*n* – Breakpoint

The BP*n* CSRs hold an instruction breakpoint address and a control bit that determines whether the breakpoint is enabled. Because instructions are always word-aligned, only the 30 most significant bits of the breakpoint address are needed. After reset, the value of the BP*n* CSRs is h00000000.

These CSRs are only implemented if DEBUG_ENABLED equals TRUE.

Figure 22: Format of the BP*n* CSRs

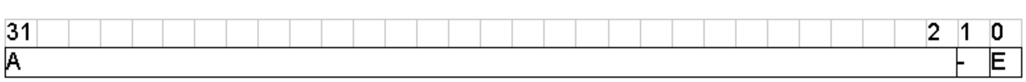


Table 16: BP*n* CSR Fields

Field	Value	Description
E	b0 – Breakpoint is disabled b1 – Breakpoint is enabled	Breakpoint enable
A		Breakpoint address (Bits 31:2)

WP*n* – Watchpoint

The WP*n* CSRs hold data watchpoint addresses. After reset, the value of the WP*n* CSRs is h00000000. These CSRs are only implemented if DEBUG_ENABLED equals TRUE.

Instruction Set Categories

LatticeMico32 supports a variety of instructions for arithmetic, logic, data comparison, data movement, and program control. Not all instructions are available in all configurations of the processor. Support for some types of instructions can be eliminated to reduce the amount of FPGA resources used. See "Configuring the LatticeMico32 Processor" on page 33.

Instructions ending with the letter "i" use an immediate value instead of a register. Instructions ending with "hi" use a 16-bit immediate and the high 16 bits from a register. Instructions ending with the letter "u" treat the data as unsigned integers.

For descriptions of individual instructions, see "Instruction Set" on page 47.

Arithmetic

The instruction set includes the standard 32-bit integer arithmetic operations. Support for the multiply and divide instructions is optional.

- Add: add, addi
- Subtract: sub
- Multiply: mul, muli
- Divide and modulus: divu, modu

There are also instructions to sign-extend byte and half-word data to word size. Support for these instructions is optional.

- Sign-extend: sextb, sexth

Logic

The instruction set includes the standard 32-bit bitwise logic operations. Most of the logic instructions also have 16-bit immediate or high 16-bit versions.

- AND: and, andi, andhi
- OR: or, ori, orhi

- Exclusive-OR: xor, xori
- Complement: not
- NOR: nor, nori
- Exclusive-NOR: xnor, xnori

Comparison

The instruction set has basic comparison instructions with versions for register-to-register and register-to-16-bit-immediate and signed and unsigned comparisons. The instructions return 1 if true and 0 if false.

- Equal: cmpe, cmpei
- Not equal: cmpne, cmpnei
- Greater: cmpg, cmpgi, cmpgu, cmpgui
- Greater or equal: cmpge, cmpgei, cmpgeu, cmpgeui

Shift

The instruction set supports left and right shifting of data in general-purpose registers. The number of bits to shift can be given through a register or a 5-bit immediate. The right shift instruction has signed and unsigned versions (also known as arithmetic and logical shifting). Support for shift instructions is optional.

- Left shift: sl, sli
- Right shift: sr, sri, sru, srui

Data Transfer

Data transfer includes instructions that move data of byte, half-word, and word sizes between memory and registers. Memory addresses are relative and given as the sum of a general-purpose register and a signed 16-bit immediate, for example, (r2+32).

- Load register from memory: lb, lbu, lh, lhu, lw

  Byte and half-word values are either sign-extended or zero-extended to fill the register.

- Store register to memory: sb, sh, sw

  Byte and half-word values are taken from the lowest order part of the register.

There are also instructions for moving data from one register to another, including general-purpose and control and status registers.

- Move between general-purpose registers: mv
- Move immediate to high 16 bits of register: mvhi
- Read and write control and status register: rcsr, wcsr

Program Flow Control

Program flow control instructions include branches, function and exception calls, and returns. The conditional branches and the immediate versions of the unconditional branch and call instructions establish the next instruction's address by adding a signed immediate to the PC register. Since the immediate is signed, the jump can be to a lower or higher address.

- Unconditional branch: b, bi
- Branch if equal: be
- Branch if not equal: bne
- Branch if greater: bg, bgu
- Branch if greater or equal: bge, bgeu
- Function call and return: call, calli, ret
- System call: scall
- Return from exception: eret
- Software breakpoint and return: break, bret

3

Configuring the LatticeMico32 Processor

This chapter describes possible configuration options that you can use for the LatticeMico32 processor. You are expected to use the Lattice Mico System Builder (MSB) tool to configure the LatticeMico32 processor. Use the processor's configuration GUI, located in the MSB, to specify the Verilog parameters of the processor's RTL. For more information on the processor's configuration GUI, refer to LatticeMico32 online Help.

Configuration Options

Table 17 describes the Verilog parameters for the LatticeMico32 processor.

Table 17: Verilog Configuration Options

Parameter Name	Values	Default	Description
MC_MULTIPLY_ENABLED	TRUE, FALSE	FALSE	Enables LUT-based multicycle multiplier. mul, muli instructions are implemented. Multiply instructions take 32 cycles to complete.
PL_MULTIPLY_ENABLED	TRUE, FALSE	TRUE	Enables pipelined multiplier (uses DSP blocks if available). mul, muli instructions are implemented. Multiply instructions take 3 cycles to complete.
DIVIDE_ENABLED	TRUE, FALSE	FALSE	Determines whether the divide and modulus instructions (divu, modu) are implemented.

Table 17: Verilog Configuration Options (Continued)

Parameter Name	Values	Default	Description
MC_BARREL_SHIFT_ENABLED	TRUE, FALSE	FALSE	Enables LUT-based multicycle barrel shifter. Enables shift instructions (sr, sri, sru, srui, sl, sli). Each shift instruction can take up to 32 cycles. If both SIGN_EXTEND_ENABLED and PL_BARREL_SHIFT_ENABLED are FALSE, this option must be set to TRUE.
PL_BARREL_SHIFT_ENABLED	TRUE, FALSE	TRUE	Enables pipelined barrel shifter. Enables shift instructions (sr, sri, sru, srui, sl, sli). Shift instructions take 3 cycles to complete. If both MC_BARREL_SHIFT_ENABLED and SIGN_EXTEND_ENABLED are FALSE, this option must be set to TRUE.
SIGN_EXTEND_ENABLED	TRUE, FALSE	FALSE	Determines whether the sign-extension instructions (sextb, sexth) are implemented. If both MC_BARREL_SHIFT_ENABLED and PL_BARREL_SHIFT_ENABLED are FALSE, this option must be set to TRUE.
DEBUG_ENABLED	TRUE, FALSE	TRUE	Determines whether software-based debugging support is implemented (that is, a ROM monitor is required to debug).
HW_DEBUG_ENABLED	TRUE, FALSE	TRUE	Determines whether hardware-based debugging support is implemented (that is, a ROM monitor is not required to debug). If this option is set to TRUE, DEBUG_ENABLED and JTAG_ENABLED must also be set to TRUE.
ROM_DEBUG_ENABLED	TRUE, FALSE	FALSE	Determines whether support for debugging ROM-based programs is implemented. If this option is set to TRUE, DEBUG_ENABLED must also be set to TRUE.
BREAKPOINTS	0-4		Specifies the number of breakpoint CSRs. If this option is set to a non-zero value, ROM_DEBUG_ENABLED must be set to TRUE.
WATCHPOINTS	0-4		Specifies the number of watchpoint CSRs. If this option is set to a non-zero value, SW_DEBUG_ENABLED must be set to TRUE.
JTAG_ENABLED	TRUE, FALSE	TRUE	Determines whether a JTAG interface is implemented.
JTAG_UART_ENABLED	TRUE, FALSE	TRUE	Determines whether a JTAG UART is implemented. If this option is set to TRUE, JTAG_ENABLED must be set to TRUE.
CYCLE_COUNTER_ENABLED	TRUE, FALSE	FALSE	Determines whether a cycle counter is implemented.
ICACHE_ENABLED	TRUE, FALSE	TRUE	Determines whether an instruction cache is implemented.

Table 17: Verilog Configuration Options (Continued)

Parameter Name	Values	Default	Description
ICACHE_BASE_ADDRESS	Any address aligned to the size of the cacheable region.	0	Specifies the base address of region cacheable by instruction cache.
ICACHE_LIMIT	Any integer multiple of the capacity of the cache added to the base address of the cacheable region	0x7FFFFFFF	Specifies the upper limit of region cacheable by instruction cache.
ICACHE_SETS	128, 256, 512, 1024	512	Specifies the number of sets in the instruction cache.
ICACHE_ASSOCIATIVITY	1, 2	1	Specifies the associativity of instruction cache.
ICACHE_BYTES_PER_LINE	4, 8, 16	4	Specifies the number of bytes per instruction cache line.
DCACHE_ENABLED	TRUE, FALSE	TRUE	Determines whether a data cache is implemented.
DCACHE_BASE_ADDRESS	Any address aligned to the size of the cacheable region	0	Specifies the base address of region cacheable by data cache.
DCACHE_LIMIT	Any integer multiple of the capacity of the cache added to the base address of the cacheable region	0x0FFFFFFF	Specifies the upper limit of region cacheable by data cache.
DCACHE_SETS	128, 256, 512, 1024	512	Specifies the number of sets in the data cache.
DCACHE_ASSOCIATIVITY	1, 2	1	Specifies the associativity of the data cache.
DCACHE_BYTES_PER_LINE	4, 8, 16	4	Specifies the number of bytes per data cache line.
INTERRUPTS	0–32	32	Specifies the number of external interrupts.
EBA_RESET	Any 256-byte aligned address	0	Specifies the reset value of the EBA CSR.
DEBA_RESET	Any 256-byte aligned address	0	Specifies the reset value of the DEB_CSR.

Table 17: Verilog Configuration Options (Continued)

Parameter Name	Values	Default	Description
EBR_POSEDGE_REGISTER_FILE	TRUE, FALSE	FALSE	Use EBR to implement register file instead of distributed RAM (LUTs).
CFG_ALTERNATE_EBA	TRUE, FALSE	FALSE	Enable dynamic switching of EBA to DEBA via "at_debug" input pin. When the "at_debug" pin is asserted (logic 1), DEBA is used. When the "at_debug" pin is deasserted (logic 0), EBA is used.

EBR Use

The following details of embedded block RAM (EBR) use with different configurations are based on the LatticeECP family of FPGAs.

- Software-based debugging (DEBUG_ENABLED) requires two EBRs.
- The instruction and data caches (ICACHE_ENABLED and DCACHE_ENABLED, respectively) require EBR based on the size of the cache:

  cache size = sets × bytes per cache line × associativity

  number of EBR = cache size/EBR_Size

For example, the default LatticeMico32 processor in the MSB has software-based debugging, an instruction cache, and a data cache. Both caches have 512 sets, 16 bytes per cache line, and an associativity of 1.

For each cache:

cache size = 512 × 16 × 1 = 8192

EBR_size = memory size when configured as a 9-bit memory.

- LatticeECP/XP = 1204x9
- LatticeECP2/XP2/ECP3 = 2048x9

number of EBR (LatticeECP/XP) = 8192/1024 + 1 = 9

Total number of EBRs required:

Software-based debugging	2
Instruction cache	9
Data cache	9
	20

4

WISHBONE Interconnect Architecture

This chapter describes the standard WISHBONE interconnect architecture that is employed by LatticeMico32 System. It focuses on the items that you must be aware of to begin designing and programming the functions of your system interconnects.

Introduction to WISHBONE Interconnect

LatticeMico32 System uses a standard WISHBONE interconnect architecture to connect the processor to its on-chip component resources, such as the LatticeMico32 UART and the LatticeMico32 SPI.

The WISHBONE interconnect works as a general-purpose interface, defining the standard data exchanges between the processor module and its components. The interconnect does not interfere with the regulation of the processor or component application-specific functions. Like microcomputer buses, the WISHBONE bus is flexible enough to be tailored to a specific application, robust enough to provide a number of bus cycles and data path widths to solve various system issues, and universal enough to allow a number of suppliers to create design products for it, making it more cost-effective.

For more information on the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, as it is formally known, refer to the OPENCORES.ORG Web site at www.opencores.org/projects.cgi/web/wishbone. The subject matter is very detailed and goes beyond the scope of this manual.

WISHBONE Registered Feedback Mode

This section describes the WISHBONE Registered Feedback mode. To implement an advanced synchronous cycle termination scheme, Registered Feedback mode bus cycles use the Cycle Type Identifier, CTI_O( ) and CTI_I( ), address tags. Both master and slave interfaces support CTI_O( ) and CTI_I( ) for improved bandwidth. The type of burst information is provided by the Burst Type Extension, BTE_O( ) and BTE_I( ) address tags.

All WISHBONE Registered Feedback-compatible cores must support WISHBONE Classic bus cycles.

Design new IP cores to support WISHBONE Registered Feedback bus cycles to ensure maximum throughput in all systems.

CTI_IO( )

The cycle-type identifier CTI_IO( ) address tag provides additional information about the current cycle. The master sends this information to the slave. The slave can use this information to prepare the response for the next cycle.

Table 18: Cycle Type Identifiers

CTI_O(2:0)	Description
000	Classic cycle
001	Constant address burst cycle
010	Incrementing burst cycle
011	Reserved
100	Reserved
101	Reserved
110	Reserved
111	End of burst

Observe the following allowances and rules:

- Master and slave interfaces may be designed to support the CTI_I( ) and CTI_O( ) signals. Also, master and slave interfaces may be designed to support a limited number of burst types.
- Master and slave interfaces that do support the CTI_I( ) and CTI_O( ) signals must at least support the Classic cycle CTI_IO( )=000 and the End-of-Cycle CTI_IO( )=111.
- Master and slave interfaces that are designed to support a limited number of burst types must complete the unsupported cycles as though they were WISHBONE Classic cycle, that is, CTI_IO( )=000.
- For description languages that allow default values for input ports (like VHDL), CTI_I( ) may be assigned a default value of 000.

- In addition to the WISHBONE Classic rules for generating cycle termination signals ACK_O, RTY_O, and ERR_O, a SLAVE may assert a termination cycle without checking the STB_I signal.
- ACK_O, RTY_O, and ERR_O may be asserted while STB_O is negated.
- A cycle terminates when the cycle termination signal, STB_I and STB_O are asserted. Even if ACK_O/ACK_I is asserted, the other signals are only valid when STB_O/STB_I is also asserted.

To avoid the inherent wait state in synchronous termination schemes, the slave must generate the response as soon as possible, that is, the next cycle. It can use the CTI_I( ) signals to determine the response for the next cycle, but if it cannot determine the state of STB_I for the next cycle, it must generate the response independent of STB_I.

BTE_IO( )

The burst-type extension BTE_IO( ) address tag provides additional information about the current burst. The master sends this information to the slave. This information is only relevant for incrementing bursts. In the future, other burst types may use these signals. See Table 19 for BTE_IO(1:0) signal incrementing and decrementing bursts.

Table 19: Burst Type Extension Signal Bursts

BTE_IO(1:0)	Description
00	Linear burst
01	4-beat wrap burst
10	8-beat wrap burst
11	16-beat wrap burst

Observe the following allowances and rules:

- Master and slave interfaces that support incrementing burst cycles must support the BTE_O( ) and BTE_I( ) signals.
- Master and slave interfaces may be designed to support a limited number of burst extensions.
- Master and slave interfaces that are designed to support a limited number of burst extensions must complete the unsupported cycles as though they were WISHBONE Classic cycle, that is, CTI_IO( )= 000.

Component Signals

In Mico System Builder (MSB), you define which components are in the platform and what needs to communicate with what. When the platform generator is run in MSB, it uses this information to build the WISHBONE-based interconnect of the platform. This generated interconnect is a set of Verilog wires connecting the various processor and component ports. To do this, the components must implement certain ports and follow a specific port-naming convention.

Table 20 defines the suffixes that must be used on the names of a component's ports. The suffixes of the ports of a master port are different than those of a slave port. The generated interconnect creates signals with names that end with the same suffix as the component port to which the signal is attached. Table 20 also notes which signals are mandatory and which are optional to support the basic WISHBONE bus cycle.

The prefixes used in the port and signal naming are not described in this section.

The port and signal descriptions that follow refer to the port or signal that ends with the string in the title.

Table 20: List of Component Port and Signal Name Suffixes

Master Ports			Slave Ports		
Name	**Width**	**Optional (O)/ Mandatory (M)**	**Name**	**Width**	**Optional (O)/ Mandatory (M)**
_ADR_O	32 bits	M	_ADR_I	32 bits	M
_DAT_O	32 bits	M	_DAT_I	32 bits	M
_DAT_I	32 bits	M	_DAT_O	32 bits	M
_SEL_O	4 bits	M	_SEL_I	4 bits	M
_WE_O	1 bit	M	_WE_I	1 bit	M
_ACK_I	1 bit	M	_ACK_O	1 bit	M
_ERR_I	1 bit	O	_ERR_O	1 bit	O
_RTY_I	1 bit	O	_RTY_O	1 bit	O
_CTI_O	3 bits	O	_CTI_I	3 bits	O
_BTE_O	2 bits	O	_BTE_I	2 bits	O
_LOCK_O	1 bit	O	_LOCK_I	1 bit	O
_CYC_O	1 bit	M	_CYC_I	1 bit	M
_STB_O	1 bit	M	_STB_I	1 bit	M

Master Port and Signal Descriptions

This section describes the master ports and signals listed in Table 20.

ADR_O [31:2]

The address output array ADR_O( ) is used to pass a binary address. ADR_O( ) actually has a full 32 bits. But, because all addressing is on DWORD (4-byte) boundaries, the lowest two bits are always zero.

DAT_O [31:0]

The data output array DAT_O( ) is used to store a binary value for output.

DAT_I [31:0]

The data input array DAT_I( ) is used to store a binary value for input.

SEL_O [3:0]

The Select Output array SEL_O( ) indicates where valid data is expected on the DAT_I( ) signal array during READ cycles and where it is placed on the DAT_O( ) signal array during WRITE cycles. The array boundaries are determined by the granularity of a port.

WE_O

The write enable output WE_O indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles and is asserted during WRITE cycles.

ACK_I

This signal is called the acknowledge input ACK_I. When asserted, the signal indicates the normal termination of a bus cycle by the slave. Also see the ERR_I and RTY_I signal descriptions.

ERR_I

The Error Input ERR_I indicates an abnormal cycle termination by the slave. The source of the error and the response generated by the master depends on the master functionality. Also see the ACK_I and RTY_I signal descriptions.

RTY_I

The Retry Input RTY_I indicates that the interface is not ready to accept or send data, so the cycle should be retried. The core functionality defines when and how the cycle is retried. Also see the ERR_I and RTY_I signal descriptions.

CTI_O [2:0]

For descriptions of the cycle-type identifier CTI_O( ), see “CTI_IO( )” on page 38.

BTE_O [1:0]

For descriptions of the burst-type extension BTE_O( ), see “BTE_IO( )” on page 39.

LOCK_O

The lock output LOCK_O, when asserted, indicates that the current bus cycle cannot be interrupted. Lock is asserted to request complete ownership of the bus. After the transfer starts, the INTERCON does not grant the bus to any other master until the current master negates LOCK_O or CYC_O.

CYC_O

The cycle output CYC_O, when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The CYC_O signal is asserted during the first data transfer and remains asserted until the last data transfer. The CYC_O signal is useful for interfaces with multi-port interfaces, such as dual-port memories. In these cases, the CYC_O signal requests the use of a common bus from an arbiter.

STB_O

The strobe output STB_O indicates a valid data transfer cycle. It is used to qualify various other signals on the interface, such as SEL_O( ). The slave asserts either the ACK_I, ERR_I, or RTY_I signals in response to every assertion of the STB_O signal.

Slave Port and Signal Descriptions

This section describes the slave ports and signals listed in the Table 20.

ADR_I [31:2]

The address input array ADR_I( ) is used to pass a binary address. ADR_I( ) actually has a full 32 bits. But, because all addressing is on DWORD (4-byte) boundaries, the lowest two bits are always zero.

DAT_I [31:0]

The data input array DAT_I( ) is used to store a binary value for input.

DAT_O [31:0]

The data output array DAT_O( ) is used to store a binary value for output.

SEL_I [3:0]

The select input array SEL_I( ) indicates where valid data is placed on the DAT_I( ) signal array during WRITE cycles and where it should be present on the DAT_O( ) signal array during READ cycles. The array boundaries are determined by the granularity of a port.

WE_I

The write enable Input WE_I indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles and is asserted during WRITE cycles.

ACK_O

The acknowledge output ACK_O, when asserted, indicates the termination of a normal bus cycle by the slave. Also see the ERR_O and RTY_O signal descriptions.

ERR_O

The error output ERR_O indicates an abnormal cycle termination by the slave. The source of the error and the response generated by the master depends on the master functionality. Also see the ACK_O and RTY_O signal descriptions.

RTY_O

The retry output RTY_O indicates that the slave interface is not ready to accept or send data, so the cycle should be retried. The core functionality defines when and how the cycle is retried. Also see the ERR_O and RTY_O signal descriptions.

CTI_I

For descriptions of the cycle-type identifier CTI_I( ), see “CTI_IO( )” on page 38.

BTE_I [1:0]

For descriptions of the burst-type extension BTE_i( ), see “BTE_IO( )” on page 39.

LOCK_I

The lock input LOCK_I, when asserted, indicates that the current bus cycle is uninterruptible. A slave that receives the LOCK LOCK_I signal is accessed by a single master only until either LOCK_I or CYC_I is negated.

CYC_I [2:0]

The Cycle Input CYC_I, when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The CYC_I signal is asserted during the first data transfer and remains asserted until the last data transfer.

STB_I

The strobe input STB_I, when asserted, indicates a valid data transfer cycle. A slave responds to other WISHBONE signals only when this STB_I is asserted, except for the RST_I signal, to which it should always respond. The slave asserts either the ACK_O, ERR_O, or RTY_O signals in response to every assertion of the STB_I signal.

Arbitration Schemes

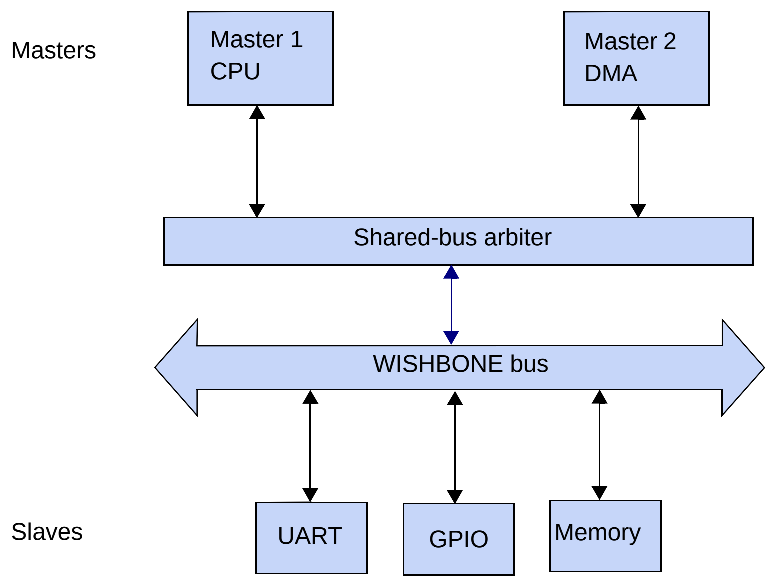
MSB supports the following arbitration schemes for platform generation:

- Shared-bus arbitration schemes
- Slave-side fixed arbitration schemes
- Slave-side round-robin arbitration schemes

Shared-Bus Arbitration

The shared-bus arbitration scheme is shown in Figure 23.

Figure 23: Bus Architecture with Shared-Bus Arbitration

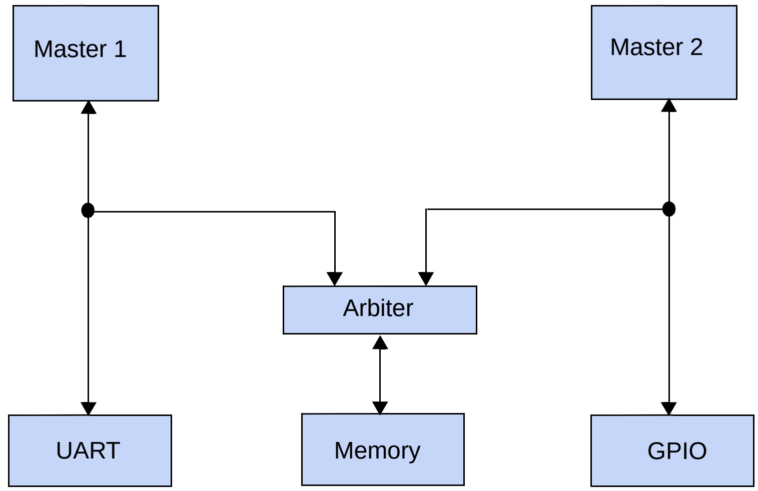


In the shared-bus arbitration scheme, one or more bus masters and bus slaves connect to a shared bus. A single arbiter controls the bus, that is, the path between masters and slaves. Each bus master requests control of the bus from the arbiter, and the arbiter grants access to a single master at a time. Once a master has control of the bus, it performs transfers with a bus slave. If multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master according to fixed arbitration rules, forcing all other masters to wait.

Slave-Side Arbitration

Slave-side arbitration is shown in Figure 24.

Figure 24: Bus Architecture with Slave-Side Arbitration



In slave-side arbitration, each multi-master slave has its own arbiter. A master port never waits to access a slave port, unless a different master port attempts to access the same slave port at the same time. As a result, multiple master ports active at the same time simultaneously transfer data with independent slave ports.

In the slave-side arbitration scheme, arbitation is only required when two or more masters contend for the same slave port. This scheme is called slave-side arbitration because it is implemented when two or more masters connect to a single slave.

Slave-Side Fixed Arbitration

In the slave-side fixed arbitration scheme, when two or more masters request control of the bus for the same slave simultaneously, the master with the highest priority gains access to the bus. At every slave transfer, only requesting masters are included in the arbitration. The master with the highest priority is granted access to the bus.

Slave-Side Round-Robin Arbitration

In the slave-side round-robin arbitration scheme, when multiple masters contend for access to a slave port, the arbiter grants access to the bus in round-robin order. At every slave transfer, only requesting masters are included in the round-robin arbitration.

5

Instruction Set

This chapter includes descriptions of all of the instruction opcodes of the LatticeMico32 processor.

Instruction Formats

All LatticeMico32 instructions are 32 bits wide. They are in four basic formats, as shown in Figure 25 through Figure 28.

Figure 25: Register Immediate (RI) Format

31	30 26	25 21	20 16	15 0
0	Op	Reg 0	Reg 1	Immediate

Figure 26: Register Register (RR) Format

31	30 26	25 21	20 16	15 11	10										0
1	Op	Reg 0	Reg 1	Reg 2	0	0	0	0	0	0	0	0	0	0	0

Figure 27: Control Register (CR) Format

31	30 26	25 21	20 16	15															0
1	Op	CSR	Reg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 28: Immediate (I) Format

31				26	25 0
1	1	1	00/11	0	Immediate

Opcode Look-Up Table

Opcode	Decimal	Hexadecimal	Mnemonic
000000	00	00	srui
000001	01	01	nori
000010	02	02	muli
000011	03	03	sh
000100	04	04	lb
000101	05	05	sri
000110	06	06	xori
000111	07	07	lh
001000	08	08	andi
001001	09	09	xnori
001010	10	0A	lw
001011	11	0B	lhu
001100	12	0C	sb
001101	13	0D	addi
001110	14	0E	ori
001111	15	0F	sli
010000	16	10	lbu
010001	17	11	be
010010	18	12	bg
010011	19	13	bge
010100	20	14	bgeu
010101	21	15	bgu
010110	22	16	sw
010111	23	17	bne
011000	24	18	andhi
011001	25	19	cmpei
011010	26	1A	cmpgi
011011	27	1B	cmpgei
011100	28	1C	cmpgeui
011101	29	1D	cmpgui
011110	30	1E	orhi
011111	31	1F	cmpnei

Opcode	Decimal	Hexadecimal	Mnemonic
100000	32	20	sru
100001	33	21	nor
100010	34	22	mul
100011	35	23	divu
100100	36	24	rcsr
100101	37	25	sr
100110	38	26	xor
100111	39	27	div
101000	40	28	and
101001	41	29	xnor
101010	42	2A	reserved
101011	43	2B	raise
101100	44	2C	sextb
101101	45	2D	add
101110	46	2E	or
101111	47	2F	sl
110000	48	30	b
110001	49	31	modu
110010	50	32	sub
110011	51	33	reserved
110100	52	34	wcsr
110101	53	35	mod
110110	54	36	call
110111	55	37	sexth
111000	56	38	bi
111001	57	39	cmpe
111010	58	3A	cmpg
111011	59	3B	cmpge
111100	60	3C	cmpgeu
111101	61	3D	cmpgu
111110	62	3E	calli
111111	63	3F	cmpne

Pseudo-Instructions

To aid the semantics of assembler programs, the LatticeMico32 assembler implements a variety of pseudo-instructions. Table 21 lists these instructions and to what actual instructions they are mapped. Disassemblers show the actual implementation.

Table 21: Pseudo-Instructions

Mnemonic	Implementation	Description
ret	b ra	Returns from function call.
mv rX, rY	or rX, rY, r0	Moves value in rY to rX.
mvhi rX, imm16	orhi rX, r0, imm16	Moves the 16-bit, left-shifted immediate into rX.
not rX, rY	xnor rX, rY, r0	Is the bitwise complement of the value in rY and stores the result in rX.
mvi	addi rd, r0, imm16	Adds 16-bit immediate to r0 and stores the result in rd. **Note:** GCC compiler tool chain expects r0 contents to be zero.
nop	addi r0, r0, 0	Adds 0 to r0 and saves it to r0, resulting in no operation (nop).

Instruction Descriptions

Some of the following tables include these parameters:

- Syntax – Describes the assembly language syntax for the instruction.
- Issue – The “issue” cycles mean the number of cycles that the microprocessor takes to place this instruction in the pipeline. For example, if the issue is 1 cycle, the next instruction will be introduced into the pipeline the very next cycle. If the issue is 4, the next instruction will be introduced three cycles later. The branches and calls are issue 4 cycles, which means that the pipeline stalls for the next three cycles.
- Semantics – Describes how the instruction creates a result from the inputs and where it puts the result. The Semantics feature refers to terms used in the assembly language syntax for the instruction.

  The Semantics feature also uses the following terms:

  - gpr – Refers to a general-purpose register.
  - PC – Refers to a program counter.
  - csr – Refers to a control and status register.
  - IE.BIE – Refers to the BIE bit of the IE (interrupt enable) register.
  - IE.IE – Refers to the IE bit of the IE (interrupt enable) register.
  - IE.EIE – Refers to the EIE bit of the IE (interrupt enable) register.

- EBA – See “EBA – Exception Base Address” on page 13.
- DEBA – See “DEBA – Debug Exception Base Address” on page 28.
- DC.RE – Refers to the RE bit of DC register. The DC register is an internal microprocessor register that is statically set to 0. It cannot be changed through the microprocessor configuration graphical user interface or parameter settings.

- Result – Specifies how many clock cycles before the result of the instruction is available. The exact result depends on the instruction. For example, for an add instruction, the result is the value produced by adding the two operands. For a load instruction, the result is the value loaded from memory.

add

Figure 29: add Instruction

31					26	25 21	20 16	15 11	10										0
1	0	1	1	0	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 22: add Instruction Features

Feature	Description
Operation	Integer addition
Description	Adds the value in rY to the value in rZ, storing the result in rX.
Syntax	`add rX, rY, rZ`
Example	`add r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] + gpr[rZ]`
Result	1 cycle
Issue	1 cycle
See Also	addi, addition with immediate

addi

Figure 30: addi Instruction

31					26	25 21	20 16	15 0
0	0	1	1	0	1	rY	rX	imm16

Table 23: addi Instruction Features

Feature	Description
Operation	Integer addition with immediate
Description	Adds the value in rY to the sign-extended immediate, storing the result in rX.

Table 23: addi Instruction Features

Feature	Description
Syntax	`addi rX, rY, imm16`
Example	`addi r4, r2, -32`
Semantics	`gpr[rX] = gpr[rY] + sign_extend(imm16)`
Result	1 cycle
Issue	1 cycle
See Also	add, addition between registers

and

Figure 31: and Instruction

31					26	25–21	20–16	15–11	10										0
1	0	1	0	0	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 24: and Instruction Features

Feature	Description
Operation	Bitwise logical AND
Description	Bitwise AND of the value in rY with the value in rZ, storing the result in rX.
Syntax	`and rX, rY, rZ`
Example	`and r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] & gpr[rZ]`
Result	1 cycle
Issue	1 cycle
See Also	andi, AND with immediate; andhi, AND with high 16 bits

andhi

Figure 32: andhi Instruction

31					26	25–21	20–16	15–0
0	1	1	0	0	0	rY	rX	imm16

Table 25: andhi Instruction Features

Feature	Description
Operation	Bitwise logical AND (high 16-bits)
Description	Bitwise AND of the value in rY with the 16-bit, left-shifted immediate, storing the result in rX.

Table 25: andhi Instruction Features

Feature	Description
Syntax	`andhi rX, rY, imm16`
Example	`andhi r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] & (imm16 << 16)`
Result	1 cycle
Issue	1 cycle
See Also	AND between registers; andi, AND with immediate

andi

Figure 33: andi Instruction

31					26	25–21	20–16	15–0
0	0	1	0	0	0	rY	rX	imm16

Table 26: andi Instruction Features

Feature	Description
Operation	Bitwise logical AND
Description	Bitwise AND of the value in rY with the zero-extended immediate, storing the result in rX.
Syntax	`andi rX, rY, imm16`
Example	`andi r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] & zero_extend(imm16)`
Result	1 cycle
Issue	1 cycle
See Also	and, AND between registers; andhi, AND with high 16 bits

b

Figure 34: b Instruction

31					26	25–21	20–0
1	1	0	0	0	0	rX	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Table 27: b Instruction Features

Feature	Description
Operation	Unconditional branch
Description	Unconditional branch to address in rX. rX cannot be r30 (ea) or r31 (ba).

Table 27: b Instruction Features

Feature	Description
Syntax	`b rX`
Example	`b r3`
Semantics	`PC = gpr[rX]`
Issue	4 cycles
See Also	bi, branch with immediate

be

Figure 35: be Instruction

31					26	25 21	20 16	15 0
0	1	0	0	0	1	rY	rX	imm16

Table 28: be Instruction Features

Feature	Description
Operation	Branch if equal
Description	Compares the value in rX with the value in rY, branching to the address given by the sum of the PC and the sign-extended immediate if the values are equal.
Syntax	`be rX, rY, imm16`
Example	`be r4, r2, label`
Semantics	`if (gpr[rX] == gpr[rY])` `  PC = PC + sign_extend(imm16 << 2)`
Issue	1 cycle (not taken), 4 cycles (taken)
See Also	bne, branch if not equal

bg

Figure 36: bg Instruction

31					26	25 21	20 16	15 0
0	1	0	0	1	0	rY	rX	imm16

Table 29: bg Instruction Features

Feature	Description
Operation	Branch if greater
Description	Compares the value in rX with the value in rY, branching to the address given by the sum of the PC and the sign-extended immediate if the value in rX is greater than the value in rY. The values in rX and rY are treated as signed integers.

Table 29: bg Instruction Features

Feature	Description
Syntax	`bg rX, rY, imm16`
Example	`bg r4, r2, label`
Semantics	`if (gpr[rX] > gpr[rY])` `  PC = PC + sign_extend(imm16 << 2)`
Issue	1 cycle (not taken), 4 cycles (taken)
See Also	bgu, branch if greater, unsigned

bge

Figure 37: bge Instruction

31					26	25–21	20–16	15–0
0	1	0	0	1	1	rY	rX	imm16

Table 30: bge Instruction Features

Feature	Description
Operation	Branch if greater or equal
Description	Compares the value in rX with the value in rY, branching to the address given by the sum of the PC and the sign-extended immediate if the value in rX is greater or equal to the value in rY. The values in rX and rY are treated as signed integers.
Syntax	`bge rX, rY, imm16`
Example	`bge r4, r2, label`
Semantics	`if (gpr[rX] >= gpr[rY])` `  PC = PC + sign_extend(imm16 << 2)`
Issue	1 cycle (not taken), 4 cycles (taken)
See Also	bgeu, branch if greater or equal, unsigned

bgeu

Figure 38: bgeu Instruction

31					26	25–21	20–16	15–0
0	1	0	1	0	0	rY	rX	imm16

Table 31: bgeu Instruction Features

Feature	Description
Operation	Branch if greater or equal, unsigned
Description	Compares the value in rX with the value in rY, branching to the address given by the sum of the PC and the sign-extended immediate if the value in rX is greater or equal to the value in rY. The values in rX and rY are treated as unsigned integers.
Syntax	`bgeu rX, rY, imm16`
Example	`bgeu r4, r2, label`
Semantics	`if (gpr[rX] >= gpr[rY])` `  PC = PC + sign_extend(imm16 << 2)`
Issue	1 cycle (not taken), 4 cycles (taken)
See Also	bge, branch if greater or equal, signed

bgu

Figure 39: bgu Instruction

31					26	25 ... 21	20 ... 16	15 ... 0
0	1	0	1	0	1	rY	rX	imm16

Table 32: bgu Instruction Features

Feature	Description
Operation	Branch if greater, unsigned
Description	Compares the value in rX with the value in rY, branching to the address given by the sum of the PC and the sign-extended immediate if the value in rX is greater than the value in rY. The values in rX and rY are treated as unsigned integers.
Syntax	`bgu rX, rY, imm16`
Example	`bgu r4, r2, label`
Semantics	`if (gpr[rX] > gpr[rY])` `  PC = PC + sign_extend(imm16 << 2)`
Issue	1 cycle (not taken), 4 cycles (taken)
See Also	bg, branch if greater, signed

bi

Figure 40: bi Instruction

31					26	25 ... 0
1	1	1	0	0	0	imm26

Table 33: bi Instruction Features

Feature	Description
Operation	Unconditional branch
Description	Unconditional branch to the address given by the sum of the PC and the sign-extended immediate.
Syntax	`bi imm26`
Example	`bi label`
Semantics	`PC = PC + sign_extend(imm26 << 2)`
Issue	4 cycles
See Also	b, branch from register

bne

Figure 41: bne Instruction

31					26	25 21	20 16	15 0
0	1	0	1	1	1	rY	rX	imm16

Table 34: bne Instruction Features

Feature	Description
Operation	Branch if not equal
Description	Compares the value in rX with the value in rY, branching to the address given by the sum of the PC and the sign-extended immediate if the values are not equal.
Syntax	`bne rX, rY, imm16`
Example	`bne r4, r2, label`
Semantics	`if (gpr[rX] != gpr[rY])` `PC = PC + sign_extend(imm16 << 2)`
Issue	1 cycle (not taken), 4 cycles (taken)
See Also	be, branch if equal

break

Figure 42: break Instruction

31																															0
1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Table 35: break Instruction Features

Feature	Description
Operation	Software breakpoint
Description	Raises a breakpoint exception.
Syntax	`break`
Example	`break`

Semantics	`gpr[ba] = PC`
	`IE.BIE = IE.IE`
	`IE.IE = 0`
	`PC = DEBA + ID * 32`

Issue	4 cycles
See Also	bret, return from breakpoint

bret

Figure 43: bret Instruction

31																															0
1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 36: bret Instruction Features

Feature	Description
Operation	Return from breakpoint
Description	Unconditional branch to the address in the breakpoint address register (ba), updating interrupt enable with value saved in breakpoint interrupt enable register.
Syntax	`bret`
Example	`bret`
Semantics	`PC = gpr[ba]` `IE.IE = IE.BIE`
Issue	4 cycles
See Also	break, breakpoint

call

Figure 44: call Instruction

31					26	25 21	20																			0
1	1	0	1	1	0	rX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 37: call Instruction Features

Feature	Description
Operation	Function call
Description	Adds 4 to the PC, storing the result in ra, then unconditionally branches to the address in rX.
Syntax	`call rX`
Example	`call r3`
Semantics	`gpr[ra] = PC + 4` `PC = gpr[rX]`
Result	1 cycle
Issue	4 cycles
See Also	calli, call with immediate; ret, return from call

calli

Figure 45: calli Instruction

31				26	25 … 0
1	1	1	1	1	0	imm26

Table 38: calli Instruction Features

Feature	Description
Operation	Function call
Description	Adds 4 to the PC, storing the result in ra, then unconditionally branches to the address given by the sum of the PC and the sign-extended immediate.
Syntax	`calli imm26`
Example	`calli label`
Semantics	`gpr[ra] = PC + 4` `PC = PC + sign_extend(imm26 << 2)`
Result	1 cycle
Issue	4 cycles
See Also	call, call from register; ret, return from call

cmpe

Figure 46: cmpe Instruction

31					26	25 … 21	20 … 16	15 … 11	10										0
1	1	1	0	0	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 39: cmpe Instruction Features

Feature	Description
Operation	Compare equal
Description	Compares the value in rY with the value in rZ, storing 1 in rX if they are equal, otherwise 0.
Syntax	`cmpe rX, rY, rZ`
Example	`cmpe r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] == gpr[rZ]`
Result	2 cycles
Issue	1 cycle
See Also	cmpei, compare equal with immediate

cmpei

Figure 47: cmpei Instruction

31					26	25–21	20–16	15–0
0	1	1	0	0	1	rY	rX	imm16

Table 40: cmpei Instruction Features

Feature	Description
Operation	Compare equal
Description	Compares the value in rY with the sign-extended immediate, storing 1 in rX if they are equal, 0 otherwise.
Syntax	`cmpei rX, rY, imm16`
Example	`cmpei r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] == sign_extend(imm16)`
Result	2 cycles
Issue	1 cycle
See Also	cmpe, compare equal between registers

cmpg

Figure 48: cmpg Instruction

31					26	25–21	20–16	15–11	10										0
1	1	1	0	1	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 41: cmpg Instruction Features

Feature	Description
Operation	Compare greater
Description	Compares the value in rY with the value in rZ, storing 1 in rX if the value in rY is greater than the value in rZ, 0 otherwise. Both operands are treated as signed integers.
Syntax	`cmpg rX, rY, rZ`
Example	`cmpg r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] > gpr[rZ]`
Result	2 cycles
Issue	1 cycle
See Also	cmpgi, compare greater with immediate; cmpgu, compare greater, unsigned; cmpgui, compare greater with immediate, unsigned

cmpgi

Figure 49: cmpgi Instruction

31					26	25–21	20–16	15–0
0	1	1	0	1	0	rY	rX	imm16

Table 42: cmpgi Instruction Features

Feature	Description
Operation	Compare greater
Description	Compares the value in rY with the sign-extended immediate, storing 1 in rX if the value in rY is greater than the immediate, 0 otherwise. Both operands are treated as signed integers.
Syntax	`cmpgi rX, rY, imm16`
Example	`cmpgi r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] > sign_extend(imm16)`
Result	2 cycles
Issue	1 cycle
See Also	cmpg, compare greater between registers; cmpgu, compare greater, unsigned; cmpgui, compare greater with immediate, unsigned

cmpge

Figure 50: cmpge Instruction

31					26	25–21	20–16	15–11	10										0
1	1	1	0	1	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 43: cmpge Instruction Features

Feature	Description
Operation	Compare greater or equal
Description	Compares the value in rY with the value in rZ, storing 1 in rX if the value in rY is greater or equal to the value in rZ, 0 otherwise. Both operands are treated as signed integers.
Syntax	`cmpge rX, rY, rZ`
Example	`cmpge r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] >= gpr[rZ]`
Result	2 cycles
Issue	1 cycle
See Also	cmpgei, compare with immediate; cmpgeu, compare, unsigned; cmpgeui, compare with immediate, unsigned

cmpgei

Figure 51: cmpgei Instruction

31					26	25–21	20–16	15–0
0	1	1	0	1	1	rY	rX	imm16

Table 44: cmpgei Instruction Features

Feature	Description
Operation	Compare greater or equal
Description	Compares the value in rY with the sign-extended immediate, storing 1 in rX if the value in rY is greater or equal to the immediate, 0 otherwise. Both operands are treated as signed integers.
Syntax	`cmpgei rX, rY, imm16`
Example	`cmpgei r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] >= sign_extend(imm16)`
Result	2 cycles
Issue	1 cycle
See Also	cmpge, compare between registers; cmpgeu, compare, unsigned; cmpgeui, compare with immediate, unsigned

cmpgeu

Figure 52: cmpgeu Instruction

31					26	25–21	20–16	15–11	10										0
1	1	1	1	0	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 45: cmpgeu Instruction Features

Feature	Description
Operation	Compare greater or equal
Description	Compares the value in rY with the value in rZ, storing 1 in rX if the value in rY is greater or equal to the value in rZ, 0 otherwise. Both operands are treated as unsigned integers.
Syntax	`cmpgeu rX, rY, rZ`
Example	`cmpgeu r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] >= gpr[rZ]`
Result	2 cycles
Issue	1 cycle
See Also	cmpge, compare between registers; cmpgei, compare with immediate; cmpgeui, compare with immediate, unsigned

cmpgeui

Figure 53: cmpgeui Instruction

31					26	25 … 21	20 … 16	15 … 0
0	1	1	1	0	0	rY	rX	imm16

Table 46: cmpgeui Instruction Features

Feature	Description
Operation	Compare greater or equal
Description	Compares the value in rY with the zero-extended immediate, storing 1 in rX if the value in rY is greater or equal to the immediate, 0 otherwise. Both operands are treated as unsigned integers.
Syntax	`cmpgeui rX, rY, imm16`
Example	`cmpgeui r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] >= zero_extend(imm16)`
Result	2 cycles
Issue	1 cycle
See Also	cmpge, compare between registers; cmpgei, compare with immediate; cmpgeu, compare, unsigned

cmpgu

Figure 54: cmpgu Instruction

31					26	25 … 21	20 … 16	15 … 11	10										0
1	1	1	1	0	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 47: cmpgu Instruction Features

Feature	Description
Operation	Compare greater unsigned
Description	Compares the value in rY with the value in rZ, storing 1 in rX if the value in rY is greater than the value in rZ, 0 otherwise. Both operands are treated as unsigned integers.
Syntax	`cmpgu rX, rY, rZ`
Example	`cmpgu r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] > gpr[rZ]`
Result	2 cycles
Issue	1 cycle
See Also	cmpg, compare greater, signed; cmpgi, compare greater with immediate; cmpgui, compare greater with immediate, unsigned

cmpgui

Figure 55: cmpgui Instruction

31					26	25–21	20–16	15–0
0	1	1	1	0	1	rY	rX	imm16

Table 48: cmpgui Instruction Features

Feature	Description
Operation	Compare greater unsigned
Description	Compares the value in rY with the zero-extended immediate, storing 1 in rX if the value in rY is greater than the immediate, 0 otherwise. Both operands are treated as unsigned integers.
Syntax	`cmpgui rX, rY, imm16`
Example	`cmpgui r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] > zero_extend(imm16)`
Result	2 cycles
Issue	1 cycle
See Also	cmpg, compare greater, signed; cmpgi, compare greater with immediate; cmpgu, compare greater, unsigned

cmpne

Figure 56: cmpne Instruction

31					26	25–21	20–16	15–11	10										0
1	1	1	1	1	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 49: cmpne Instruction Features

Feature	Description
Operation	Compare not equal
Description	Compares the value in rY with the value in rZ, storing 1 in rX if they are not equal, 0 otherwise.
Syntax	`cmpne rX, rY, rZ`
Example	`cmpne r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] != gpr[rZ]`
Result	2 cycles
Issue	1 cycle
See Also	cmpnei, compare not equal with immediate

cmpnei

Figure 57: cmpnei Instruction

31					26	25–21	20–16	15–0
0	1	1	1	1	1	rY	rX	imm16

Table 50: cmpnei Instruction Features

Feature	Description
Operation	Compare not equal
Description	Compares the value in rY with the sign-extended immediate, storing 1 in rX if they are not equal, 0 otherwise.
Syntax	`cmpnei rX, rY, imm16`
Example	`cmpnei r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] != sign_extend(imm16)`
Result	2 cycles
Issue	1 cycle
See Also	cmpne, compare not equal between registers

divu

Figure 58: divu Instruction

31					26	25–21	20–16	15–11	10										0
1	0	0	0	1	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 51: divu Instruction Features

Feature	Description
Operation	Unsigned iinteger division
Description	Divides the value in rY by the value in rZ, storing the quotient in rX. Both operands are treated as unsigned integers. Available only if the processor was configured with the DIVIDE_ENABLED option.
Syntax	`divu rX, rY, rZ`
Example	`divu r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] / gpr[rZ]`
Result	34 cycles
Issue	34 cycles
See Also	modu, modulus

eret

Figure 59: eret Instruction

31																															0
1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 52: eret Instruction Features

Feature	Description
Operation	Return from exception
Description	Unconditionally branches to the address in the exception address register (ea), updating interrupt enable with value saved in exception interrupt enable register.
Syntax	`eret`
Example	`eret`
Semantics	`PC = gpr[ea]` `IE.IE = IE.EIE`
Result	
Issue	3 cycles
See Also	scall, system call

lb

Figure 60: lb Instruction

31					26	25 21	20 16	15 0
0	0	0	1	0	0	rY	rX	imm16

Table 53: lb Instruction Features

Feature	Description
Operation	Load byte from memory
Description	Loads a byte from memory at the address specified by the sum of the value in rY added to the sign-extended immediate, storing the sign-extended result into rX.
Syntax	`lb rX, (rY+imm16)`
Example	`lb r4, (r2+5)`
Semantics	`address = gpr[rY] + sign_extend(imm16)` `gpr[rX] = sign_extend(memory[address])`
Result	3 cycles
Issue	1 cycle
See Also	lbu, load byte, unsigned; lh, load half-word, signed; lhu, load half-word, unsigned; lw, load word

lbu

Figure 61: lbu Instruction

31					26	25 – 21	20 – 16	15 – 0
0	1	0	0	0	0	rY	rX	imm16

Table 54: lbu Instruction Features

Feature	Description
Operation	Load unsigned byte from memory
Description	Loads a byte from memory at the address specified by the sum of the value in rY added to the sign-extended immediate, storing the zero-extended result into rX.
Syntax	`lbu rX, (rY+imm16)`
Example	`lbu r4, (r2+5)`
Semantics	`address = gpr[rY] + sign_extend(imm16)` `gpr[rX] = zero_extend(memory[address])`
Result	3 cycles
Issue	1 cycle
See Also	lb, load byte, signed; lh, load half-word, signed; lhu, load half-word, unsigned; lw, load word

lh

Figure 62: lh Instruction

31					26	25 – 21	20 – 16	15 – 0
0	0	0	1	1	1	rY	rX	imm16

Table 55: lh Instruction Features

Feature	Description
Operation	Load half-word from memory
Description	Loads a half-word from memory at the address specified by the sum of the value in rY added to the sign-extended immediate, storing the sign-extended result into rX.
Syntax	`lh rX, (rY+imm16)`
Example	`lh r4, (r2+6)`

Semantics	`address = gpr[rY] + sign_extend(imm16)`
	`gpr[rX] = sign_extend((memory[address] << 8)`
	`| (memory[address+1]))`

Result	3 cycles
Issue	1 cycle
See Also	lb, load byte, signed; lbu, load byte, unsigned; lhu, load half-word, unsigned; lw, load word

lhu

Figure 63: lhu Instruction

31					26	25–21	20–16	15–0
0	0	1	0	1	1	rY	rX	imm16

Table 56: lhu Instruction Features

Feature	Description
Operation	Load unsigned half-word from memory
Description	Loads a half-word from memory at the address specified by the sum of the value in rY added to the sign-extended immediate, storing the zero-extended result into rX.
Syntax	`lhu rX, (rY+imm16)`
Example	`lhu r4, (r2+6)`

Semantics	`address = gpr[rY] + sign_extend(imm16)`
	`gpr[rX] = zero_extend((memory[address] << 8)`
	`| (memory[address+1]))`

Result	3 cycles
Issue	1 cycle
See Also	lb, load byte, signed; lbu, load byte, unsigned; lh, load half-word, signed; lw, load word

lw

Figure 64: lw Instruction

31					26	25–21	20–16	15–0
0	0	1	0	1	0	rY	rX	imm16

Table 57: lw Instruction Features

Feature	Description
Operation	Load word from memory
Description	Loads a word from memory at address specified by the sum of the value in rY added to the sign-extended immediate, storing the result in rX.
Syntax	`lw rX, (rY+imm16)`
Example	`lw r4, (r2+8)`

Semantics	`address = gpr[rY] + sign_extend(imm16)`
	`gpr[rX] = (memory[address] << 24)`
	`        | (memory[address+1] << 16)`
	`        | (memory[address+2] << 8)`
	`        | (memory[address+3])`

Result	3 cycles
Issue	1 cycle
See Also	lb, load byte, signed; lbu, load byte, unsigned; lh, load half-word, signed; lhu, load half-word, unsigned

modu

Figure 65: modu Instruction

31					26	25–21	20–16	15–11	10										0
1	1	0	0	0	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 58: modu Instruction Features

Feature	Description
Operation	Unsigned integer modulus
Description	Divides the value in rY by the value in rZ, storing the remainder in rX. Both operands are treated as unsigned integers. Available only if the processor was configured with the DIVIDE_ENABLED option.
Syntax	`modu rX, rY, rZ`
Example	`modu r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] % gpr[rZ]`
Result	34 cycles
Issue	34 cycles
See Also	divu, divide

mul

Figure 66: mul Instruction

31					26	25–21	20–16	15–11	10										0
1	0	0	0	1	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 59: mul Instruction Features

Feature	Description
Operation	Integer multiply
Description	Multiplies the value in rY by the value in rZ, storing the low 32 bits of the product in rX. Available only if the processor was configured with either the MC_MULTIPLY_ENABLED or PL_MULTIPLY_ENABLED option.
Syntax	`mul rX, rY, rZ`
Example	`mul r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] * gpr[rZ]`
Result	3 cycles
Issue	1 cycle
See Also	muli, multiply with immediate

muli

Figure 67: muli Instruction

31					26	25–21	20–16	15–0
0	0	0	0	1	0	rY	rX	imm16

Table 60: muli Instruction Features

Feature	Description
Operation	Integer multiply
Description	Multiplies the value in rY by the sign-extended immediate, storing the low 32 bits of the product in rX. Available only if the processor was configured with either the MC_MULTIPLY_ENABLED or PL_MULTIPLY_ENABLED option.
Syntax	`muli rX, rY, imm16`
Example	`muli r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] * sign_extend(imm16)`
Result	3 cycles
Issue	1 cycle
See Also	mul, multiply between registers

mv

Feature	Description
Operation	Move
Description	Moves the value in rY to rX. This is a pseudo-instruction implemented with: `or rX, rY, r0`.
Syntax	`mv rX, rY`
Example	`mv r4, r2`
Semantics	`gpr[rX] = gpr[rY] | gpr[r0]`
Result	1 cycle
Issue	1 cycle
See Also	mvhi, move immediate into high 16 bits

mvhi

Feature	Description
Operation	Move high 16 bits
Description	Moves the 16-bit, left-shifted immediate into rX. This is a pseudo-instruction implemented with: `orhi rX, r0, imm16`.
Syntax	`mvhi rX, imm16`
Example	`mvhi r4, 0x5555`
Semantics	`gpr[rX] = gpr[r0] | (imm16 << 16)`
Result	1 cycle
Issue	1 cycle
See Also	mv, move between registers

nor

Figure 68: nor Instruction

31 26	25 21	20 16	15 11	10 0
1 0 0 0 0 1	rY	rZ	rX	0 0 0 0 0 0 0 0 0 0 0

Table 61: nor Instruction Features

Feature	Description
Operation	Bitwise logical NOR
Description	Bitwise NOR of the value in rY with the value in rZ, storing the result in rX.
Syntax	`nor rX, rY, rZ`
Example	`nor r14, r15, r17`
Semantics	`gpr[rX] = ~(gpr[rY] | gpr[rZ])`
Result	1 cycle
Issue	1 cycle
See Also	nori, NOR with immediate

nori

Figure 69: nori Instruction

31 26	25 21	20 16	15 0
0 0 0 0 0 1	rY	rX	imm16

Table 62: nori Instruction Features

Feature	Description
Operation	Bitwise logical NOR
Description	Bitwise NOR of the value in rY with the zero-extended immediate, storing the result in rX.
Syntax	`nori rX, rY, imm16`
Example	`nori r4, r2, 0x5555`
Semantics	`gpr[rX] = ~(gpr[rY] | zero_extend(imm16))`
Result	1 cycle
Issue	1 cycle
See Also	nor, NOR between registers

not

Feature	Description
Operation	Bitwise complement
Description	Bitwise complement of the value in rY, storing the result in rX. This is a pseudo-instruction implemented with: `xnor rX, rY, r0`.
Syntax	`not rX, rY`
Example	`not r4, r2`
Semantics	`gpr[rX] = ~(gpr[rY] ^ gpr[r0])`
Result	1 cycle
Issue	1 cycle

or

Figure 70: or Instruction

31					26	25 21	20 16	15 11	10										0
1	0	1	1	1	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 63: or Instruction Features

Feature	Description
Operation	Bitwise logical OR
Description	Bitwise OR of the value in rY with the value in rZ, storing the result in rX.
Syntax	`or rX, rY, rZ`
Example	`or r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] | gpr[rZ]`
Result	1 cycle
Issue	1 cycle
See Also	ori, OR with immediate; orhi, OR with high 16 bits

ori

Figure 71: ori Instruction

31					26	25 – 21	20 – 16	15 – 0
0	0	1	1	1	0	rY	rX	imm16

Table 64: ori Instruction Features

Feature	Description
Operation	Bitwise logical OR
Description	Bitwise OR of the value in rY with the zero-extended immediate, storing the result in rX.
Syntax	`ori rX, rY, imm16`
Example	`ori r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] | zero_extend(imm16)`
Result	1 cycle
Issue	1 cycle
See Also	or, OR between registers; orhi, OR with high 16 bits

orhi

Figure 72: orhi Instruction

31					26	25 – 21	20 – 16	15 – 0
0	1	1	1	1	0	rY	rX	imm16

Table 65: orhi Instruction Features

Feature	Description
Operation	Bitwise logical OR (high 16-bits)
Description	Bitwise OR of the value in rY with the 16-bit, left-shifted immediate, storing the result in rX.
Syntax	`orhi rX, rY, imm16`
Example	`orhi r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] | (imm16 << 16)`
Result	1 cycle
Issue	1 cycle
See Also	or, OR between registers; ori, OR with immediate

rcsr

Figure 73: rcsr Instruction

31					26	25–21	20				16	15–11											0
1	0	0	1	0	0	csr	0	0	0	0	0	rX	0	0	0	0	0	0	0	0	0	0	0

Table 66: rcsr Instruction Features

Feature	Description
Operation	Read control and status register
Description	Reads the value of the specified control and status register and stores it in rX.
Syntax	`rcsr rX, csr`
Example	`rcsr r15, IM`
Semantics	`gpr[rX] = csr`
Result	1 cycle
Issue	1 cycle
See Also	wcsr, write control and status register

ret

Feature	Description
Operation	Return from function call
Description	Unconditional branch to address in ra. This is a pseudo-instruction implemented with: `b ra`.
Syntax	`ret`
Example	`ret`
Semantics	`PC = gpr[ra]`
Result	
Issue	4 cycle**s**
See Also	call, function call from register; calli, function call with immediate

sb

Figure 74: sb Instruction

31					26	25–21	20–16	15–0
0	0	1	1	0	0	rX	rY	imm16

Table 67: sb Instruction Features

Feature	Description
Operation	Store byte to memory
Description	Stores the lower byte in rY into memory at the address specified by the sum of the value in rX added to the sign-extended immediate.
Syntax	`sb(rX+imm16), rY`
Example	`sb(r2+8), r4`
Semantics	`address = gpr[rX] + sign_extend(imm16)` `memory[address] = gpr[rY] & 0xff`
Result	
Issue	1 cycle
See Also	sh, store half-word; sw, store word

scall

Figure 75: scall Instruction

31																															0
1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Table 68: scall Instruction Features

Feature	Description
Operation	System call
Description	Raises a system call exception.
Syntax	`scall`
Example	`scall`

Semantics	`gpr[ea] = PC`
	`IE.EIE = IE.IE`
	`IE.IE = 0`
	`PC = (DC.RE ? DEBA : EBA) + ID * 32`

Result	
Issue	4 cycles
See Also	eret, return from exception

sextb

Figure 76: sextb Instruction

31					26	25 … 21	20				16	15 … 11	10										0
1	0	1	1	0	0	rY	0	0	0	0	0	rX	0	0	0	0	0	0	0	0	0	0	0

Table 69: sextb Instruction Features

Feature	Description
Operation	Sign-extend byte to word
Description	Sign-extends the value in rY, storing the result in rX. Available only if the processor was configured with the SIGN_EXTEND_ENABLED option.
Syntax	`sextb rX, rY`
Example	`sextb r14, r15`
Semantics	`gpr[rX] = (gpr[rY] << 24) >> 24`
Result	1 cycle
Issue	1 cycle
See Also	sexth, sign-extend half-word

sexth

Figure 77: sexth Instruction

31					26	25–21	20				16	15–11	10										0
1	1	0	1	1	1	rY	0	0	0	0	0	rX	0	0	0	0	0	0	0	0	0	0	0

Table 70: sexth Instruction Features

Feature	Description
Operation	Sign-extends half-word to word
Description	Sign-extends the value in rY, storing the result in rX. Available only if the processor was configured with the SIGN_EXTEND_ENABLED option.
Syntax	`sexth rX, rY`
Example	`sexth r14, r15`
Semantics	`gpr[rX] = (gpr[rY] << 16) >> 16`
Result	1 cycle
Issue	1 cycle
See Also	sextb, sign-extend byte

sh

Figure 78: sh Instruction

31					26	25–21	20–16	15–0
0	0	0	0	1	1	rX	rY	imm16

Table 71: sh Instruction Features

Feature	Description
Operation	Store half-word to memory
Description	Stores the lower half-word in rY into memory at the address specified by the sum of the value in rX added to the sign-extended immediate.
Syntax	`sh (rX+imm16), rY`
Example	`sh (r2+8), r4`

Semantics	`address = gpr[rX] + sign_extend(imm16)`
	`memory[address] = (gpr[rY] >> 8) & 0xff`
	`memory[address+1] = gpr[rY] & 0xff`

Result	
Issue	1 cycle
See Also	sb, store byte; sw, store word

sl

Figure 79: sl Instruction

31					26	25–21	20–16	15–11	10										0
1	0	1	1	1	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 72: sl Instruction Features

Feature	Description
Operation	Shift left
Description	Shifts the value in rY left by the number of bits specified by the value in rZ, storing the result in rX. Available only if the processor was configured with either the MC_BARREL_SHIFT_ENABLED or PL_BARREL_SHIFT_ENABLED option.
Syntax	`sl rX, rY, rZ`
Example	`sl r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] << (gpr[rZ] & 0x1f)`
Result	2 cycles
Issue	1 cycle
See Also	sli, shift left with immediate

sli

Figure 80: sli Instruction

31					26	25–21	20–16	15											–0
0	0	1	1	1	1	rY	rX	0	0	0	0	0	0	0	0	0	0	0	imm5

Table 73: sli Instruction Features

Feature	Description
Operation	Shift left
Description	Shifts the value in rY left by the number of bits specified by the immediate, storing the result in rX. Available only if the processor was configured with either the MC_BARREL_SHIFT_ENABLED or PL_BARREL_SHIFT_ENABLED option.
Syntax	`sli rX, rY, imm5`
Example	`sli r4, r2, 17`
Semantics	`gpr[rX] = gpr[rY] << imm5`
Result	2 cycles
Issue	1 cycle
See Also	sl, shift left from register

sr

Figure 81: sr Instruction

31					26	25–21	20–16	15–11	10										0
1	0	0	1	0	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 74: sr Instruction Features

Feature	Description
Operation	Shift right (arithmetic)
Description	Shifts the signed value in rY right by the number of bits specified by the value in rZ, storing the result in rX. Available only if the processor was configured with either the MC_BARREL_SHIFT_ENABLED or PL_BARREL_SHIFT_ENABLED option.
Syntax	`sr rX, rY, rZ`
Example	`sr r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] >> (gpr[rZ] & 0x1f)`
Result	2 cycles
Issue	1 cycle
See Also	sri, shift right with immediate; sru, shift right, unsigned; srui, shift right with immediate, unsigned

sri

Figure 82: sri Instruction

31					26	25–21	20–16	15											4–0
0	0	0	1	0	1	rY	rX	0	0	0	0	0	0	0	0	0	0	0	imm5

Table 75: sri Instruction Features

Feature	Description
Operation	Shift right (arithmetic)
Description	Shifts the signed value in rY right by the number of bits specified by the immediate, storing the result in rX. Available only if the processor was configured with either the MC_BARREL_SHIFT_ENABLED or PL_BARREL_SHIFT_ENABLED option.
Syntax	`sri rX, rY, imm5`
Example	`sri r4, r2, 12`
Semantics	`gpr[rX] = gpr[rY] >> imm5`
Result	2 cycles
Issue	1 cycle
See Also	sr, shift right from register; sru, shift right, unsigned; srui, shift right with immediate, unsigned

sru

Figure 83: sru Instruction

31					26	25–21	20–16	15–11	10										0
1	0	0	0	0	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 76: sru Instruction Features

Feature	Description
Operation	Shift right, unsigned (logical)
Description	Shifts the unsigned value in rY right by the number of bits specified by the value in rZ, storing the result in rX. Available only if the processor was configured with either the MC_BARREL_SHIFT_ENABLED or PL_BARREL_SHIFT_ENABLED option.
Syntax	`sru rX, rY, rZ`
Example	`sru r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] >> (gpr[rZ] & 0x1f)`
Result	2 cycles
Issue	1 cycle
See Also	sr, shift right from register; sri, shift right with immediate; srui, shift right with immediate, unsigned

srui

Figure 84: srui Instruction

31					26	25 21	20 16	15											0
0	0	0	0	0	0	rY	rX	0	0	0	0	0	0	0	0	0	0	0	imm5

Table 77: srui Instruction Features

Feature	Description
Operation	Shifts right, unsigned (logical)
Description	Shifts the unsigned value in rY right by the number of bits specified by the immediate, storing the result in rX. Available only if the processor was configured with either the MC_BARREL_SHIFT_ENABLED or PL_BARREL_SHIFT_ENABLED option.
Syntax	`srui rX, rY, imm5`
Example	`srui r4, r2, 5`
Semantics	`gpr[rX] = gpr[rY] >> imm5`
Result	2 cycles
Issue	1 cycle
See Also	sr, shift right from register; sri, shift right with immediate; sru, shift right, unsigned

sub

Figure 85: sub Instruction

31					26	25 21	20 16	15 11	10										0
1	1	0	0	1	0	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 78: sub Instruction Features

Feature	Description
Operation	Integer subtraction
Description	Subtracts the value in rZ from the value in rY, storing the result in rX.
Syntax	`sub rX, rY, rZ`
Example	`sub r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] - gpr[rZ]`
Result	1 cycle
Issue	1 cycle
See Also	addi, add with signed immediate

sw

Figure 86: sw Instruction

31					26	25–21	20–16	15–0
0	1	0	1	1	0	rX	rY	imm16

Table 79: sw Instruction Features

Feature	Description
Operation	Store word to memory
Description	Stores the value in rY into memory at the address specified by the sum of the value in rX added to the sign-extended immediate.
Syntax	`sw(rX+imm16), rY`
Example	`sw(r2+8), r4`

Semantics	`address = gpr[rX] + sign_extend(imm16)`
	`memory[address] = (gpr[rY] >>24) & 0xff`
	`memory[address+1] = (gpr[rY] >>16) & 0xff`
	`memory[address+2] = (gpr[rY] >>8) & 0xff`
	`memory[address+3] = gpr[rY] & 0xff`

Result	
Issue	1 cycle
See Also	sb, store byte; sh, store half-word

wcsr

Figure 87: wcsr Instruction

31					26	25–21	20–16	15															0
1	1	0	1	0	0	csr	rX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 80: wcsr Instruction Features

Feature	Description
Operation	Write control or status register
Description	Writes the value in rX to the specified control or status register.
Syntax	`wcsr csr, rX`
Example	`wcsr IM, r15`
Semantics	`csr = gpr[rX]`
Result	1 cycle
Issue	1 cycle
See Also	rcsr, read control and status register

xnor

Figure 88: xnor Instruction

31					26	25 21	20 16	15 11	10										0
1	0	1	0	0	1	rY	rZ	rX	0	0	0	0	0	0	0	0	0	0	0

Table 81: xnor Instruction Features

Feature	Description
Operation	Bitwise logical exclusive-NOR
Description	Bitwise exclusive-NOR of the value in rY with the value in rZ, storing the result in rX.
Syntax	`xnor rX, rY, rZ`
Example	`xnor r14, r15, r17`
Semantics	`gpr[rX] = ~(gpr[rY] ^ gpr[rZ])`
Result	1 cycle
Issue	1 cycle
See Also	xnori, XNOR with immediate

xnori

Figure 89: xnori Instruction

31					26	25 21	20 16	15 0
0	0	1	0	0	1	rY	rX	imm16

Table 82: xnori Instruction Features

Feature	Description
Operation	Bitwise logical exclusive-NOR
Description	Bitwise exclusive-NOR of the value in rY with the zero-extended immediate, storing the result in rX.
Syntax	`xnori rX, rY, imm16`
Example	`xnori r4, r2, 0x5555`
Semantics	`gpr[rX] = ~(gpr[rY] ^ zero_extend(imm16))`
Result	1 cycle
Issue	1 cycle
See Also	xnor, XNOR between registers

xor

Figure 90: xor Instruction

31–26	25–21	20–16	15–11	10–0
1 0 0 1 1 0	rY	rZ	rX	0 0 0 0 0 0 0 0 0 0 0

Table 83: xor Instruction Features

Feature	Description
Operation	Bitwise logical exclusive-OR
Description	Bitwise exclusive-OR of the value in rY with the value in rZ, storing the result in rX.
Syntax	`xor rX, rY, rZ`
Example	`xor r14, r15, r17`
Semantics	`gpr[rX] = gpr[rY] ^ gpr[rZ]`
Result	1 cycle
Issue	1 cycle
See Also	xori, XOR with immediate

xori

Figure 91: xori Instruction

31–26	25–21	20–16	15–0
0 0 0 1 1 0	rY	rX	imm16

Table 84: xori Instruction Features

Feature	Description
Operation	Bitwise logical exclusive-OR
Description	Bitwise exclusive-OR of the value in rY with the zero-extended immediate, storing the result in rX.
Syntax	`xori rX, rY, imm16`
Example	`xori r4, r2, 0x5555`
Semantics	`gpr[rX] = gpr[rY] ^ zero_extend(imm16)`
Result	1 cycle
Issue	1 cycle
See Also	xori, XOR between registers

Index