LatticeMico8 Processor Reference Manual

December 2012

Copyright

Trademarks

Disclaimers

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

Introduction

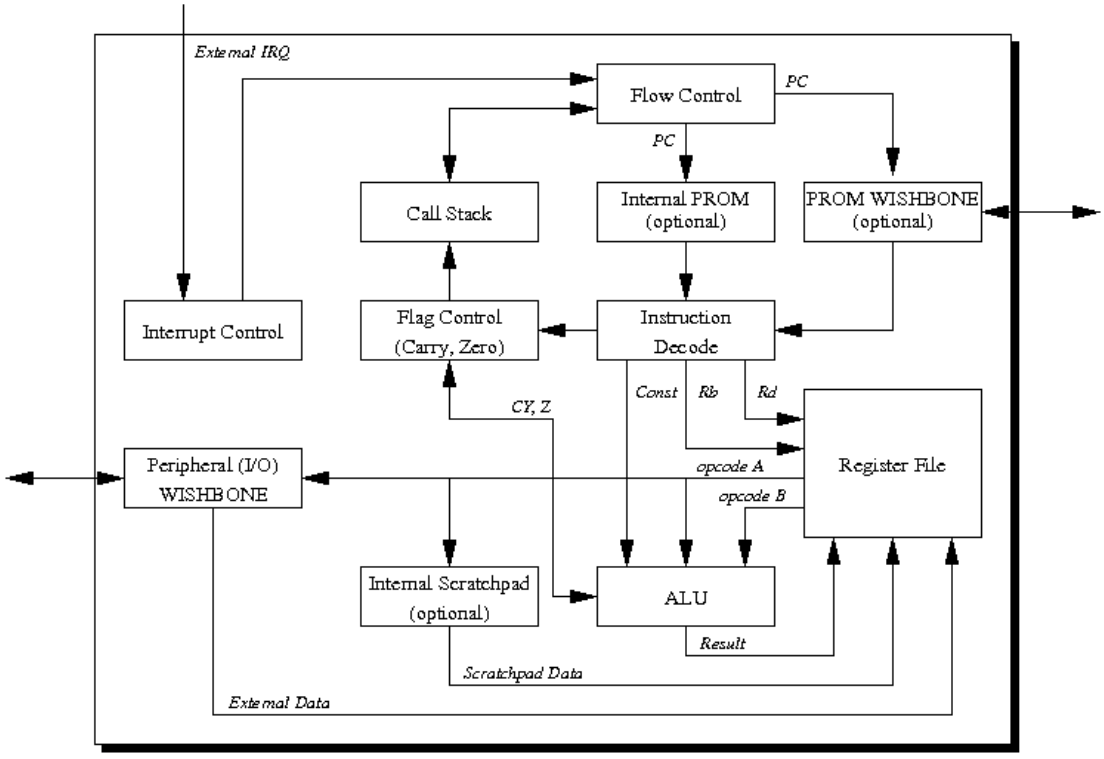
The LatticeMico8™ is an 8-bit microcontroller optimized for Field Programmable Gate Arrays (FPGAs) and Programmable Logic Device architectures from Lattice Semiconductor. It combines a full 18-bit wide instruction set with 16 or 32 general-purpose registers. It is suitable for a wide variety of markets, including communications, consumer, computer, medical, industrial and automotive. The core consumes minimal device resources—fewer than 250 Look-Up Tables (LUTs) in the smallest configuration—while maintaining a broad feature set.

LatticeMico8 Features

- 8-Bit Data Path
- 18-Bit Instructions
- Configurable Instruction Memory (PROM)
  - Internal, or external through the WISHBONE Interface
  - Configurable to accommodate 256, 512, 1K, 1.5K, 2K, 2.5K, 3K, 3.5K or 4K instructions
- Scratchpad Memory
  - Internal, or external through the WISHBONE Interface
  - Configurable up to 4Gbytes using paged bytes (256 bytes/page)
- Input/Output Peripheral Space through the WISHBONE Interface
  - Configurable up to 4Gbytes using paged ports (256 ports/page)
- Minimum Two Cycles per Instruction
- Configurable 16 or 32 General-purpose Registers
- Configurable Call Stack size

Figure 1 on page 2 shows the LatticeMico8 Microcontroller block diagram.

Figure 1: LatticeMico8 Microcontroller Core



Architecture

This chapter describes the LatticeMico8 register and memory architecture and explains the interrupt architecture and call stack.

Register Architecture

This section describes the general-purpose and control and status registers of the LatticeMico8 architecture.

General-Purpose Registers

The LatticeMico8 microcontroller can be configured to have either 16 or 32 general-purpose registers. Each register is 8 bits wide. The registers are implemented using a dual-port distributed memory. The LatticeMico8 opcode set permits the microcontroller to access 32 registers. When LatticeMico8 is configured with 16 registers, any opcode reference to R16 to R31 maps to R0 to R15 respectively.

General-purpose registers R13, R14, and R15 can also be used by the LatticeMico8 microcontroller as page-pointer registers, depending on the current memory mode. Page pointers (PP) are used when the scratchpad and peripheral memory spaces are larger than 256 bytes (see "Memory Modes" on page 9). The memory address is formed by concatenating the values in registers R13, R14, and R15 with an 8-bit value derived from the LatticeMico8 memory instruction. Table 1 on page 4 highlights the three LatticeMico8 memory modes and corresponding designation of registers R13, R14, and R15.

- In the large memory mode, registers R13, R14, and R15 indicate which of the 16M pages is currently active. R13 provides the least-significant byte of page address and R15 provides most-significant byte.

- In the medium memory mode, register R13 indicates which of the 256 pages is currently active.

Table 1: Designation of LatticeMico8 Registers Based on LatticeMico8 Memory Mode

	LatticeMico8 Memory Mode		
Register Number	**Small**	**Medium**	**Large**
0 through 12	general-purpose	general-purpose	general-purpose
13	general-purpose	PP	PP (LSB)
14	general-purpose	general-purpose	PP
15	general-purpose	general-purpose	PP (MSB)
16 through 31	general-purpose	general-purpose	general-purpose

Control and Status Registers

Table 2 shows all the names of the control and status registers (CSR), the read and write access, and the index used when the register is accessed. All signal levels are active high.

Table 2: Control and Status Registers

Name	Access	Index	Description
IP	R/W	0	Interrupt Pending
IM	R/W	1	Interrupt Mask
IE	R/W	2	Global Interrupt Enable/Disable

IP – Interrupt Pending The IP CSR contains a pending bit for each of the 8 external 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 0.

IM – Interrupt Mask The IM CSR contains an enable bit for each of the 8 external 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 0.

IE – Global Interrupt Enable The IE CSR contains a single-bit (bit position 0) flag, IE, which determines whether interrupts are enabled. This flag has priority over the IM CSR. After reset, the value of the IE CSR is 0.

Memory Architecture

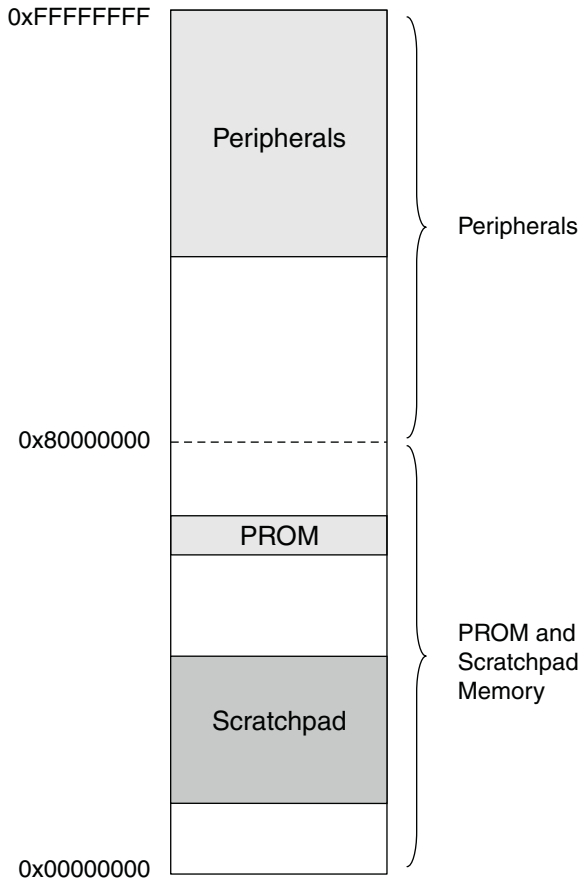
This section describes the memory architecture of the LatticeMico8 microcontroller.

Memory Regions

The LatticeMico8 microcontroller recognizes three independent memory regions. Each memory region has its own independent input/output interface and its own instruction set support. These three memory regions are called the PROM, the Scratchpad, and the Peripheral memory regions respectively. The size and location of each of these memory regions is configurable as long as all these three memory regions are located entirely within the 4GB address space. These memory regions can also be configured to overlap within LatticeMico System Builder. Figure 2 shows the three memory regions and the address space to which they are confined by LatticeMico System Builder.

See “Acessing LatticeMico8 Memory Regions” on page 42 for details on how to access each of the three memory regions from a software programmer's perspective.

Figure 2: Memory Organization



PROM Space

The PROM memory region contains the program code that will be executed by the LatticeMico8 microcontroller core and is accessible via its instruction fetch engine. The size of the PROM memory region can be configured to accommodate 256, 512, 1024, 2048, or 4096 instruction opcodes. By default the memory region is located within the LatticeMico8 microcontroller. The memory regions can also be configured to be external to the LatticeMico8 microcontroller.

When the PROM memory region is internal to the microcontroller, it is connected to the LatticeMico8 instruction fetch engine via a dedicated high-speed bus that fetches one instruction opcode per clock cycle. There is no instruction set support to write to internal PROM. When the PROM memory region is external to the microcontroller, it is accessed by the master WISHBONE interface within the LatticeMico8 instruction fetch engine. This WISHBONE interface has a 8-bit data bus and it takes three 8-bit WISHBONE accesses to fetch one LatticeMico8 instruction opcode. The instruction fetch latency is now dictated by the system WISHBONE latency and the latency of the PROM memory. The minimum instruction fetch latency is 12 clock cycles. Table 3 shows the WISHBONE interface signals. For more information about 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.

Table 3: PROM WISHBONE Interface Signals

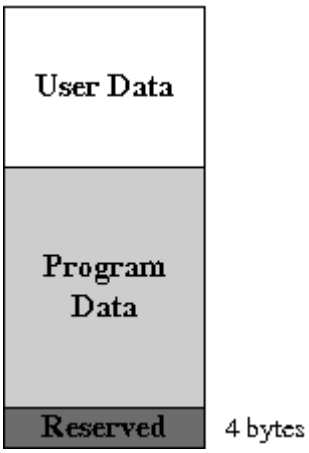
Name	Width	Direction	Description
I_CYC_O	1	Output	A new LatticeMico8 instruction fetch request is initiated by asserting this signal. This signal remains asserted until I_ACK_I is asserted, which indicates the completion of the request.
I_STB_O	1	Output	A new LatticeMico8 instruction fetch request is initiated by asserting this signal. This signal may be valid only for the first cycle.
I_CTI_O	2	Output	Always has a value 2'b00
I_BTE_O	3	Output	Always has a value 3'b000
I_ADR_O	32	Output	The address output array I_ADR_O( ) is used to pass a binary address.
I_WE_O	1	Output	Always has a value 1'b0
I_SEL_O	4	Output	Always has a value 4'b1111
I_DAT_O	8	Output	Unused
I_LOCK_O	1	Output	Unused (signal exists, but it is not implemented)
I_ACK_I	1	Input	When asserted, the signal indicates the normal termination of a bus cycle and that an instruction is available on I_DAT_I bus.
I_ERR_I	1	Input	Unused (signal exists, but it is not implemented)
I_RTY_I	1	Input	Unused (signal exists, but it is not implemented)
I_DAT_I	8	Input	One byte of the LatticeMico8 18-bit instruction opcode is available on this bus when I_ACK_I is asserted. It takes three WISHBONE transactions to complete one LatticeMico8 instruction fetch.

The advantage of configuring the PROM memory region as external to the LatticeMico8 microcontroller is that the PROM memory region can now be configured to overlap with other LatticeMico8 memory regions within Lattice Mico System Builder and, therefore, be directly written to by LatticeMico8 opcodes. This configuration also offers the ability to store and execute LatticeMico8 instructions from non-volatile memory such as Flash. As shown in Figure 2 on page 5, the external PROM memory region can be placed at any location within a 4GB address range. When the LatticeMico8 microcontroller is instantiated using Lattice Mico System Builder, it will restrict the placement of external PROM between 0x00000000 and 0x80000000.

Scratchpad Space

LatticeMico8 provides an independent memory space that is designed to be used for program read/write and read-only data as well as other user-defined data. The size of this scratchpad memory can be configured from 32 bytes to 4G bytes, in power-of-two increments. Figure 3 shows the structure of this scratchpad space and how data is located within this space. The scratchpad memory space can be placed at any location within a 4GB address range. The first 4 bytes are reserved for LatticeMico8 interrupt handling. Program data is situated above this reserved space. The designer can configure the size of scratchpad memory that is used for program data. User-defined data is optional and is always located after program data.

Figure 3: Scratchpad Space Structure



The scratchpad memory can be configured to be entirely internal to the LatticeMico8 microcontroller, entirely external to LatticeMico8 microcontroller, or a combination of both.

- The internal scratchpad is implemented using single-port EBRs and is hooked up to the LatticeMico8 core through a dedicated bus. Reads or writes to the internal scratchpad take a single clock cycle.
- The external scratchpad is accessed through the Peripheral WISHONE interface of the LatticeMico8 microcontroller (see “Interrupt Architecture” on page 10). Each read or write will take a minimum of 2 clock cycles.

Peripheral (Input/Output) Space

LatticeMico8 provides an independent memory space that is designed to be used for peripherals and other memory-mapped hardware. The size of this peripheral memory space can be configured from 0 bytes to 4G bytes in power-of-two increments. While the peripheral memory space can be placed at any location within a 4GB address range, Lattice Mico System Builder restricts the peripheral memory space to the addresses between 0x80000000 and 0xFFFFFFFF.

This memory space is always external to the LatticeMico8 microcontroller and is primarily used to enable LatticeMico8 to communicate with memory-mapped hardware and peripherals. The LatticeMico8 microcontroller can communicate with any hardware or peripheral within the peripheral memory space, through the peripheral WISHBONE interface within LatticeMico8 core, using LatticeMico8 instruction opcodes. This WISHBONE interface has 8-bit input and output data busses and a 32-bit address bus. Table 4 shows the Peripheral WISHBONE interface signals.

Table 4: Peripheral WISHBONE Interface Signals

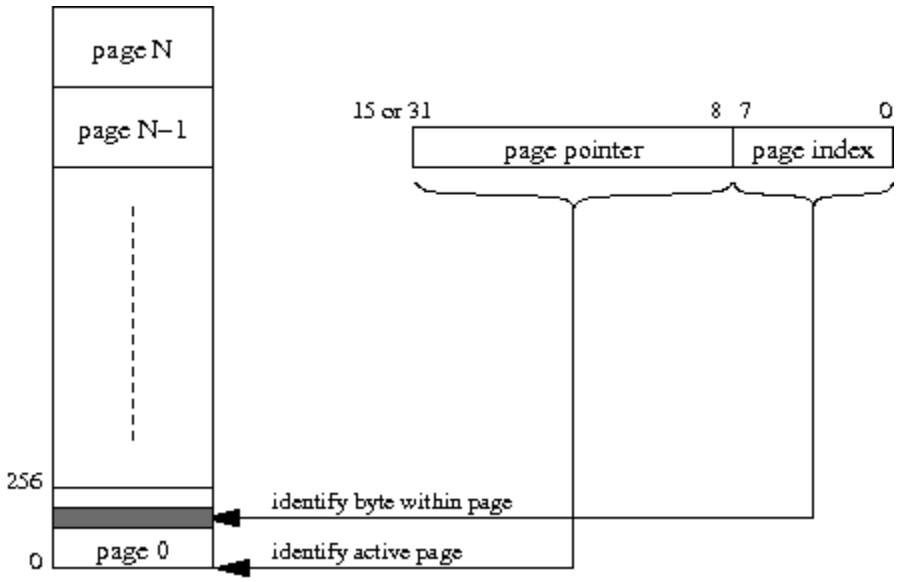
Name	Width	Direction	Description
D_CYC_O	1	Output	A new LatticeMico8 data request is initiated by asserting this signal. This signal remains asserted until D_ACK_I is asserted, which indicates completion of the request.
D_STB_O	1	Output	A new LatticeMico8 data request is initiated by asserting this signal. This signal may be valid only for first cycle.
D_CTI_O	2	Output	This bus will always have a value 2'b00
D_BTE_O	3	Output	This bus will always have a value 3'b000
D_ADR_O	32	Output	The address output array D_ADR_O( ) is used to pass a binary address. D_ADR_O( ) actually has a full 32 bits.
D_WE_O	1	Output	This signal indicates whether a new data request is a read (0) or a write (1). This signal must hold its value as long as D_CYC_O is asserted.
D_SEL_O	1	Output	Always has a value 1'b1
D_DAT_O	8	Output	Has valid data when D_WE_O is 1'b1.
D_LOCK_O	1	Output	Unused (signal exists, but it is not implemented)
D_ACK_I	1	Input	When asserted, the signal indicates the normal termination of a bus cycle.
D_ERR_I	1	Input	Unused (signal exists, but it is not implemented)
D_RTY_I	1	Input	Unused (signal exists, but it is not implemented)
D_DAT_I	8	Input	Data is available on this bus when D_ACK_I and D_WEO are asserted.

Memory Modes

The LatticeMico8 microcontroller can be configured for different sizes for the scratchpad and peripheral memory regions. The size of scratchpad and peripheral memory regions can be as small as 32 bytes and as large as 4G bytes. A 32-byte memory region requires only 5 address bits, while a 4GB memory region requires 32 address bits.

The LatticeMico8 instruction set can directly access only 256 memory locations, since all general-purpose registers are 8 bits wide. (See “Instruction Set” on page 13.) To access memory regions that are larger than 256 bytes, LatticeMico8 relies on a concept called “paging,” in which the memory is logically divided into 256-byte pages. The memory address is composed of two parts, as shown in Figure 4: the page index and the page pointer. The page index is 8 bits wide and addresses a byte in the currently active page, while the page pointer provides the address of the currently active page.

Figure 4: Memory Modes



The page pointers are essentially general-purpose registers that have been retargeted to provide a memory address. (See “Memory Regions” on page 5.) Table 5 shows the memory modes of the LatticeMico8 microcontroller, the size of addressable memory space in each mode, and the general-purpose registers used as page pointers.

Table 5: LatticeMico8 Memory Modes

Memory Mode	Maximum Memory Size	Address Bits	Page Pointer Registers
Small	256 bytes	8	N/A
Medium	16K bytes	16	R13
Large	4G bytes	32	R13, R14, R15

Interrupt Architecture

The LatticeMico8 microcontroller supports up to 8 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. The software can query the status of the interrupts and acknowledge them through the IP CSR. If more interrupt sources or more sophisticated interrupt detection methods are required, external interrupt controllers can be cascaded onto the microcontroller's interrupt pins to provide the needed functionality.

When an interrupt is received, the address of the next instruction is pushed into the call stack (see "Call Stack" on page 10), and the microcontroller continues execution from the interrupt vector (address 0). The flags (carry and zero) are pushed onto the call stack along with the return address. An *iret* instruction will pop the call stack and transfer control to the address on top of the stack. The flags (carry and zero) are also popped from the call stack.

See "Interrupt Convention" on page 41 for details on the programming model for interrupts.

Note

The LatticeMico8 microcontroller does not support nested interrupts. Locations 0 through 3 in the scratchpad are reserved for interrupt handling and should not used for any other purpose.

Call Stack

The LatticeMico8 microcontroller implements a hardware call stack to handle procedure calls (*call* instruction) and procedure/interrupt return (*ret* and *iret* instructions). The depth of this call stack determines the number of nested procedure calls that can be handled by the LatticeMico8 microcontroller, and designers can choose the depth to be 8, 16, or 32. When a *call* instruction is executed, the address of the next instruction is pushed on to the call stack. A *ret* or *iret* instruction will pop the stack and continue execution from the location at the top of the stack.

Note

There is no mechanism in hardware to detect whether the number of nested procedure calls has exceeded the depth of the call stack. It is up to the software developer to ensure that the call stack does not overflow.

Configuration Options

The LatticeMico8 microcontroller is reconfigurable. Table 6 outlines the various configuration options that are available to a designer.

Table 6: LatticeMico8 Configuration Options

Parameter Name	Description
LATTICE_FAMILY	The target Lattice FPGA family.
CFG_PROM_INIT_FILE	Provides the file that contains the initialization data (program code) for an internal PROM.
CFG_PROM_INIT_FILE_FORMAT	Indicates whether CFG_PROM_INIT_FILE is in hex (default) or binary.
CFG_PROM_SIZE	Indicates the number of instructions that can be accommodated in the PROM.
CFG_SP_INIT_FILE	Provides the file that contains the initialization data (program data) for an internal scratchpad.
CFG_SP_INIT_FILE_FORMAT	Indicates whether CFG_SP_INIT_FILE_FORMAT is hex (default) or binary.
SP_PORT_ENABLE	Indicates whether the scratchpad is internal (value 1) or external (value 0). The default is 1.
SP_SIZE	Indicates the number of bytes in the scratchpad.
SP_BASE_ADDRESS	Provides the base address of the scratchpad, regardless of whether it is internal or external.
CFG_IO_BASE_ADDRESS	Provides the base address of the peripheral memory region.
CFG_EXT_SIZE_[8|16|32]	Indicates the size of address bus for the scratchpad and peripheral memory regions and, therefore, identifies the LatticeMico8 memory mode. The default is 16 (medium memory mode).
CFG_REGISTER_[16|32]	Indicates the number of general-purpose registers in LatticeMico8. The default is 8.

Table 6: LatticeMico8 Configuration Options (Continued)

Parameter Name	Description
CFG_CALL_STACK_[8|16|32]	Indicates the depth of the call stack. The default is 16.
CFG_ROM_EN	Indicates whether the PROM and Scratchpad memories need to be initialized from non-volatile storage such as flash at power-up. The default is 0, i.e., no copying is required.
CFG_ROM_BASE_ADDRESS	Provides the base address of the memory which contains the PROM and Scratchpad images. The PROM image starts at this base address. The Scratchpad image starts at location (CFG_PROM_SIZE*3).
CFG_XIP	Indicates whether the PROM memory is the same as the non-volatile storage that contains the PROM image. The default is 0, i.e., both memories are different. 1 indicates that both memories are the same (i.e., no copying needs to be done) and the PROM is external to LatticeMico8.
INTERRUPTS	Indicates the number of external interrupts. The default is 8.

Instruction Set

This chapter includes descriptions of all the instruction opcodes of the LatticeMico8 microcontroller.

Instruction Formats

All LatticeMico8 instructions are 18 bits wide. They are in three basic formats, as shown in Figure 5, Figure 6, and Figure 7.

Figure 5: Register-Register Format

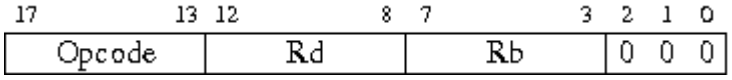


Figure 6: Register-Immediate Format

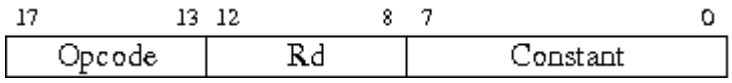
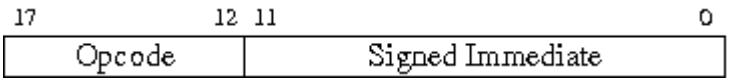


Figure 7: Immediate Format



Instruction Set Lookup Table

Table 7: Instruction Set Reference Card

Operation	Action	Flags
ADD Rd, Rb	Rd = Rd + Rb	Carry, Zero
ADDC Rd, Rb	Rd = Rd + Rb + Carry	Carry, Zero
ADDI Rd, C	Rd = Rd + C	Carry, Zero
ADDIC Rd, C	Rd = Rd + C + Carry	Carry, Zero
AND Rd, Rb	Rd = Rd & Rb	Zero
ANDI Rd, C	Rd = Rd & C	Zero
B Label	PC = PC + Label	
BC Label	If Carry = 1, PC = PC + Label	
BNC Label	If Carry = 0, PC = PC + Label	
BNZ Label	If Zero = 0, PC = PC + Label	
BZ Label	If Zero = 1, PC = PC + Label	
CALL Label	Stack = PC + 1, PC = PC + Label	
CALLC Label	If Carry = 1, Stack = PC + 1, PC = PC + Label	
CALLNC Label	If Carry = 0, Stack = PC + 1, PC = PC + Label	
CALLNZ Label	If Zero = 0, Stack = PC + 1, PC = PC + Label	
CALLZ Label	If Zero = 1, Stack = PC + 1, PC = PC + Label	
CLRC	Carry = 0	Carry
CLRI	IE = 0	
CLRZ	Zero = 0	Zero
CMP Rd, Rb	Rd – Rb	Carry, Zero
CMPI Rd, C	Rd – C	Carry, Zero
EXPORT Rd, Port#	Peripheral (Port #) = Rd	
EXPORTI Rd, Rb	Peripheral (Page Pointer, Rb) = Rd	
IMPORT Rd, Port#	Rd = Peripheral (Port #)	
IMPORTI Rd, Rb	Rd = Peripheral (Page Pointer, Rb)	
IRET	PC, Carry, Zero = Stack	Carry, Zero
LSP RD, SS	Rd = Scratchpad (SS)	
LSPI Rd, Rb	Rd = Scratchpad (Page Pointer, Rb)	
MOV Rd, Rb	Rd = Rb	
MOVI Rd, C	Rd = Const	

Table 7: Instruction Set Reference Card (Continued)

Operation	Action	Flags
NOP	PC = PC + 1	
OR Rd, Rb	Rd = Rd | Rb	Zero
ORI Rd, C	Rd = Rd | C	Zero
RCSR Rd, CRb	Rd = CSR (Rb)	
RET	PC = Stack	
ROL Rd, Rb	Rd = {(Rb<<1), Rb[0]}	Zero
ROLC Rd, Rb	Rd = {(Rb<<1), Carry}, Carry = Rb[7]	Carry, Zero
ROR Rd, Rb	Rd = {Rb[0], (Rb>>1)}	Zero
RORC Rd, Rb	Rd = {Carry, (Rb>>1)}, Carry = Rb[0]	Carry, Zero
SETC	Carry = 1	Carry
SETI	IE = 0	
SETZ	Zero = 1	Zero
SSP Rd, SS	Scratchpad (SS) = Rd	
SSPI Rd, Rb	Scratchpad (Page Pointer, Rb) = Rd	
SUB Rd, Rb	Rd = Rd – Rb	Carry, Zero
SUBC Rd, Rb	Rd = Rd – Rb – Carry	Carry, Zero
SUBI Rd, C	Rd = Rd – C	Carry, Zero
SUBIC Rd, C	Rd = Rd – C – Carry	Carry, Zero
TEST Rd, Rb	Rd & Rb	Zero
TESTI Rd, C	Rd & C	Zero
XOR Rd, Rb	Rd = Rd ^ Rb	Zero
XORI Rd, C	Rd = Rd ^ C	Zero
WCSR CRd, Rb	CSR (Rd) = Rb	Zero

Instruction Descriptions

This section describes the operations of the instruction set.

ADD Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd + Rb (add registers)

The carry flag is updated with the carry out from the addition. The zero flag is set to 1 if all the bits of the result are 0.

ADDC Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd + Rb + Carry Flag (add registers and carry flag)

The carry flag is updated with the carry out from the addition. The zero flag is set to 1 if all the bits of the result are 0.

ADDI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd + CCCCCCCC (add constant to register)

The carry flag is updated with the carry out from the addition. The zero flag is set to 1 if all the bits of the result are 0.

ADDIC Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd + CCCCCCCC + Carry Flag (add register, constant and carry flag)

The carry flag is updated with the carry out from the addition. The zero flag is set to 1 if all the bits of the result are 0.

AND Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
No	Yes

Rd = Rd and Rb (bitwise AND registers)

The zero flag is set to 1 if all the bits of the result are 0.

ANDI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
No	Yes

Rd = Rd and CCCCCCCC (bitwise AND register with constant)

The zero flag is set to 1 if all the bits of the result are 0.

B Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

Unconditional Branch. PC = PC + Signed Offset of Label

Unconditional branch. PC is incremented by the signed offset of the label from the current PC. The offset can be +2047/-2048.

BC Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Carry Flag = 1 then PC = PC + (Signed Offset of Label). Else PC = PC + 1.

Branch if carry. If carry flag is set, the PC is incremented by the signed offset of the label from the current PC. If carry flag is not set, then execution continues with the following instruction. The offset can be +2047/-2048.

BNC Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Carry Flag = 0 then PC = PC + (Signed Offset of Label). Else PC = PC + 1.

Branch if not carry. If carry flag is not set, the PC is incremented by the signed offset of the label from the current PC. If carry flag is set, then execution continues with the following instruction. The offset can be +2047/-2048.

BNZ Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Zero Flag = 0 then PC = PC + (Signed Offset of Label). Else PC = PC + 1.

Branch if not 0. If zero flag is not set, the PC is incremented by the signed offset of the label from the current PC. If zero flag is set, then execution continues with the following instruction. The offset can be +2047/-2048.

BZ Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Zero Flag = 1 then PC = PC + (Signed Offset of Label). Else PC = PC + 1.

Branch if 0. If zero flag is set, the PC is incremented by the signed offset of the label from the current PC. If zero flag is 0, then execution continues with the following instruction. The offset can be +2047/-2048.

CALL Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

Push PC + 1/C/Z into Call Stack
PC = PC + Signed offset of LABEL

Unconditional call. Address of the next instruction (PC + 1) is pushed into the call stack, and the PC is incremented by the signed offset (+2047/-2048) of the label from the current PC.

CALLC Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Carry Flag = 1, then
  Push PC + 1/C/Z into Call Stack
  PC = PC + Signed Offset of LABEL
Else, PC = PC + 1

CALL if carry. If the carry flag is set, the address of the next instruction (PC + 1) is pushed into the call stack and the PC is incremented by the signed offset (+2047/-2048) of the label from the current PC. If the carry flag is not set, then execution continues from the following instruction.

CALLNC Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Carry Flag = 0, then
  Push PC + 1/C/Z into Call Stack
  PC = PC Signed Offset of LABEL
Else, PC = PC + 1

CALL if not carry. If the carry flag is set, the address of the next instruction (PC + 1) is pushed into the call stack, and the PC is incremented by the signed offset (+2047/-2048) of the label from the current PC. If the carry flag is not set, then execution continues from the following instruction.

CALLNZ Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Zero Flag = 0, then
  Push PC + 1/C/Z into Call Stack
  PC = PC + Signed Offset of LABEL
Else PC = PC + 1

CALL if NOT 0. If the zero flag is not set, the address of the next instruction (PC + 1) is pushed into the call stack and the PC is incremented by the signed offset (+2047/-2048) of the label from the current PC. If the zero flag is set, then execution continues from the following instruction.

CALLZ Label

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	L	L	L	L	L	L	L	L	L	L	L	L

CY Flag Updated	Zero Flag Updated
No	No

If Zero Flag = 1, then
  Push PC + 1/C/Z into Call Stack
  PC = PC + Signed Offset of LABEL
Else, PC = PC + 1

CALL if 0. If the zero flag is set, the address of the next instruction (PC + 1) is pushed into the call stack and the PC is incremented by the signed offset (+2047/-2048) of the label from the current PC. If zero flag is not set, then execution continues from the following instruction.

CLRC

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	No

Carry Flag = 0

Clear carry flag.

CLRI

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0

CY Flag Updated	Zero Flag Updated
No	No

Interrupt Enable Flag = 0

Clear interrupt enable flag. Disable interrupts.

CLRZ

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

CY Flag Updated	Zero Flag Updated
No	Yes

Zero Flag = 0

Clear zero flag.

CMP Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	Yes

Subtract Rb from Rd and update the flags. The result of the subtraction is not written back.

The carry flag is set to 1 if the result is negative. The zero flag is set to 1 if all the bits of the result are 0.

CMPI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
Yes	Yes

Subtract Constant from Rd and update the flags. The result of the subtraction is not written back.

The carry flag is set to 1 if the result is negative. The zero flag is set to 1 if all the bits of the result are 0.

EXPORT Rd, Port#

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	P	P	P	P	P	0	0	0

CY Flag Updated	Zero Flag Updated
No	No

Peripheral (Port #) = Rd
Output value of Register Rd to Peripheral Address. Peripheral Address can be 0-31.

EXPORTI Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	1	0

CY Flag Updated	Zero Flag Updated
No	No

Port Value (Rb) = Rd
Indirect write to peripheral address. The peripheral address is formed by concatenating the page pointer value with the value in register Rb. In small memory mode, the peripheral address can be 0 - 255. In medium memory mode, the peripheral address can be 0 - 64K. In large memory mode, the peripheral address can be 0 - 4Gbyte.

IMPORT Rd, Port#

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	P	P	P	P	P	0	0	1

CY Flag Updated	Zero Flag Updated
No	No

Rd = Peripheral (Port #)
Read value from Peripheral (I/O) address and write in to register Rd. The Peripheral address can be 0 - 31.

IMPORTI Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	1	1

CY Flag Updated	Zero Flag Updated
No	No

Rd = Peripheral (Page Pointer + Rb)
Indirect read from peripheral address. The peripheral address is formed by concatenating the page pointer value with the value in register Rb. In small memory mode, the peripheral address can be 0 - 255. In medium memory mode, the peripheral address can be 0 - 64K. In large memory mode, the peripheral address can be 0 - 4Gbyte.

IRET

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	Yes

PC = Top of Call Stack
Pop Call Stack
Restore Zero and Carry Flags from Call Stack

Return from interrupt. In addition to popping the call stack, the carry and zero flags are restored from shadow locations.

LSP RD, SS

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	S	S	S	S	S	1	0	1

CY Flag Updated	Zero Flag Updated
No	No

Rd = Scratch Pad (SS)

Load from scratch pad memory direct. Load the value from the scratch pad location designated by constant SS into Register Rd. SS can be 0-31.

LSPI Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	1	1	1

CY Flag Updated	Zero Flag Updated
No	No

Rd = Scratch Pad (Page Pointer + Rb)

Indirect read from scratchpad address. The scratchpad address is formed by concatenating the page pointer value with the value in register Rb. In small memory mode, the peripheral address can be 0 - 255. In medium memory mode, the peripheral address can be 0 - 64K. In large memory mode, the peripheral address can be 0 - 4Gbyte.

MOV Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
No	No

Rd = Rb (move register to register)

The zero flag is set to 1 if all the bits of the result are 0.

MOVI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
No	No

Rd = CCCCCCCC (move constant into register)

The zero flag is set to 1 if all the bits of the result are 0.

NOP

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CY Flag Updated	Zero Flag Updated
No	No

PC = PC + 1
No operation moves R0 to R0.

OR Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
No	Yes

Rd = Rd | Rb (bitwise OR registers)

The zero flag is set to 1 if all the bits of the result are 0.

ORI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
No	Yes

Rd = Rd | CCCCCCCC (bitwise OR register with constant)

The zero flag is set to 1 if all the bits of the result are 0.

RET

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

CY Flag Updated	Zero Flag Updated
No	No

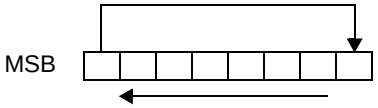
PC = Top of Call Stack
Pop Call Stack

Unconditional return. PC is set to the value on the top of the call stack.

ROL Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	1

CY Flag Updated	Zero Flag Updated
No	Yes

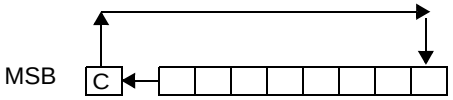


Rotate left. Register Rb is shifted left by one bit. The highest order bit is shifted into the lowest order bit. The result is written back to register Rd. The zero flag is set to 1 if all the bits of the result are 0.

ROLC Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	1	1

CY Flag Updated	Zero Flag Updated
Yes	Yes

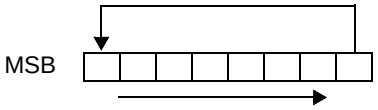


Rotate left through carry. Register Rb is shifted left by one bit. The carry flag is shifted into the lowest order bit and the highest order bit is shifted into the carry flag. The result is written back to Register Rd. The zero flag is set to 1 if all the bits of the result are 0.

ROR Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
No	Yes

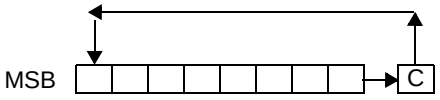


Rotate right. Register Rd is shifted right one bit and the highest order bit is replaced with the lowest order bit. The result is written back to Register Rd. The zero flag is set to 1 if all the bits of the result are 0.

RORC Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	1	0

CY Flag Updated	Zero Flag Updated
Yes	Yes



Rotate right through carry. the contents of Register Rb are shifted right one bit, the carry flag is shifted into the highest order bit, and the lowest order bit is shifted into the carry flag. The result is written back to Register Rd. The zero flag is set to 1 if all the bits of the result are 0.

SETC

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

CY Flag Updated	Zero Flag Updated
Yes	No

Carry Flag = 1

Set carry flag.

SETI

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1

CY Flag Updated	Zero Flag Updated
No	No

Interrupt Enable Flag = 1

Set interrupt enable flag. Enable interrupt.

SETZ

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1

CY Flag Updated	Zero Flag Updated
No	Yes

Zero Flag = 1

Set zero flag.

SSP Rd, SS

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	S	S	S	S	S	1	0	0

CY Flag Updated	Zero Flag Updated
No	No

Scratch Pad (SS) = Rd

Store into scratch pad memory direct. Store value of Register Rd into scratch pad memory location designated by constant SS. The location address can be 0-31.

SSPI Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	1	1	0

CY Flag Updated	Zero Flag Updated
No	No

Scratch Pad (Page Pointer + Rb) = Rd

Indirect write to scratchpad address. The scratchpad address is formed by concatenating the page pointer value with the value in register Rb. In small memory mode, the peripheral address can be 0 - 255. In medium memory mode, the peripheral address can be 0 - 64K. In large memory mode, the peripheral address can be 0 - 4Gbyte.

SUB Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd - Rb (subtract register from register)

The carry flag is set to 1 if the result is negative. The zero flag is set to 1 if all the bits of the result are 0.

SUBC Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd - Rb - Carry Flag (subtract register with carry from register)

The carry flag is set to 1 if the result is negative. The zero flag is set to 1 if all the bits of the result are 0.

SUBI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd - CCCCCCCC (subtract constant from register)

The carry flag is set to 1 if the result is negative. The zero flag is set to 1 if all the bits of the result are 0.

SUBIC Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
Yes	Yes

Rd = Rd - CCCCCCCC - Carry Flag (subtract constant with carry from register)

The carry flag is set to 1 if the result is negative. The zero flag is set to 1 if all the bits of the result are 0.

TEST Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
No	Yes

Perform a bitwise AND between Rd and Rb, update the zero flag. The result of the AND operation is not written back.

The zero flag is set to 1 if all the bits of the result are 0.

TESTI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
No	Yes

Perform a bitwise AND between Rd and Constant, update the zero flag. The result of the AND operation is not written back.

The zero flag is set to 1 if all the bits of the result are 0.

XOR Rd, Rb

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	Rd	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb	Rb	0	0	0

CY Flag Updated	Zero Flag Updated
No	Yes

Rd = Rd and Rb (bitwise XOR registers)

The zero flag is set to 1 if all the bits of the result are 0.

XORI Rd, C

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	Rd	Rd	Rd	Rd	Rd	C	C	C	C	C	C	C	C

CY Flag Updated	Zero Flag Updated
No	Yes

Rd = Rd and CC (bitwise XOR register with constant)

The zero flag is set to 1 if all the bits of the result are 0.

Programming Model

This chapter describes the LatticeMico8 programming model, including data types, calling sequence, and interrupt convention.

Data Representation

The LatticeMico8 microcontroller supports the data types listed in Table 8.

Table 8: LatticeMico8 Data Types

		Size in Memory Model			Alignment in Memory Model		
Type	**C Type**	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
Integer	Signed char	1	1	1	1	1	1
Integer	Unsigned char	1	1	1	1	1	1
Integer	Signed short	2	2	2	2	2	2
Integer	Unsigned short	2	2	2	2	2	2
Integer	Signed int	2	2	2	2	2	2
Integer	Unsigned int	2	2	2	2	2	2
Integer	Signed long	4	4	4	4	4	4
Integer	Unsigned long	4	4	4	4	4	4
Integer	Unsigned long long	4	4	4	4	4	4
Pointer	Any-type*	1	2	4	1	2	4
Floating-Point	Float	4	4	4	4	4	4

Table 8: LatticeMico8 Data Types

		Size in Memory Model			Alignment in Memory Model		
Type	**C Type**	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
Floating-Point	Double	4	4	4	4	4	4
Floating-Point	Long double	4	4	4	4	4	4

*A NULL pointer of any type must be zero. All floating-point types are IEEE-754 compliant.

Procedure Caller-Callee Convention

This section describes the standard function calling sequence, including stack frame layout, register usage, and parameter passing. The standard calling sequence requirements apply only to global functions; however, it is recommended that all functions use the standard calling sequence.

Register Usage

The register usage model shown in Table 9 is used by the LatticeMico8 Compiler. It must be used by developers who are writing ASM code that will be compiled into an executable using the LatticeMico8 compiler.

Table 9: Register Usage (SP – Stack Pointer, FP – Frame Pointer, PP – Page Pointer)

	Preserved Across Functions			Usage		
Register	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
R0	N	N	N	Arg 0/Return 0	Arg 0/Return 0	Arg 0/Return 0
R1	N	N	N	Arg 1/Return 1	Arg 1/Return 1	Arg 1/Return 1
R2	N	N	N	Arg 2/Return 2	Arg 2/Return 2	Arg 2/Return 2
R3	N	N	N	Arg 3/Return 3	Arg 3/Return 3	Arg 3/Return 3
R4	N	N	N	Arg 4	Arg 4	Arg 4
R5	N	N	N	Arg 5	Arg 5	Arg 5
R6	N	N	N	Arg 6	Arg 6	Arg 6
R7	N	N	N	Arg 7	Arg 7	Arg 7
R8	Y	Y	Y		Fixed – SP	
R9	Y	Y	Y		Fixed – SP	
R10	N	Y	N		Fixed – FP	
R11	N	Y	N		Fixed – FP	
R12	N	N	N			

Table 9: Register Usage (SP – Stack Pointer, FP – Frame Pointer, PP – Page Pointer) (Continued)

	Preserved Across Functions			Usage		
Register	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
R13	N	N	N		Fixed – PP	Fixed – PP
R14	Y	Y	N	Fixed – SP		Fixed – PP
R15	Y	N	N	Fixed – FP		Fixed – PP
R16	Y	Y	Y			
R17	Y	Y	Y			
R18	Y	Y	Y			
R19	Y	Y	Y			
R20	Y	Y	N			
R21	Y	Y	N			
R22	Y	Y	N			
R23	Y	Y	N			
R24	Y	Y	Y			Fixed – SP
R25	Y	Y	Y			Fixed – SP
R26	Y	Y	Y			Fixed – SP
R27	Y	Y	Y			Fixed – SP
R28	N	N	Y			Fixed – FP
R29	N	N	Y			Fixed – FP
R30	N	N	Y			Fixed – FP
R31	N	N	Y			Fixed – FP

Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 10 shows the stack frame organization.

The stack pointer always points to the end of the latest allocated stack frame. All frames must be aligned. The first 32 bytes below the stack frame are reserved for leaf functions that do not need to modify the stack pointer. Interrupt handlers must guarantee that they will not use this area.

Table 10: Stack Frame Layout

FP-relative Position	SP-relative Position	Contents	Frame
FP + (M – 6) . . . FP + 0	SP + (N + M + 4) . . . SP + (N + 4)	Function Argument Byte M . . . Function Argument Byte 6	Previous
FP – 1 FP – 2 FP – 3 FP – 4	SP + (N + 3) SP + (N + 2) SP + (N + 1) SP + N	Previous FP (byte 3) Previous FP (byte 2) Previous FP (byte 1) Previous FP (byte 0)	Current
FP – 5 . . . FP – (N + 5)	SP + (N – 1) . . . SP + 0	Local Variable N . . . Local Variable 0	
FP – (N + 6) . . . FP – (N + 37)	SP – 1 . . . SP – 32	Red Zone Area – Start Red Zone Area – End	Future

Parameter Passing

Functions receive their first 8 argument bytes in function argument registers R0-R7. If there are more than eight argument bytes, the remaining argument bytes are passed on the stack. Small structure and union arguments are passed in argument registers; other structure and union arguments are passed as pointers. A function that returns an integral or pointer value puts its result in the registers R0-R3. Void functions leave registers R0-R3 undefined. A function that returns a small structure or union places the returned value in registers R0-R3. Other structures and unions are returned in memory, pointed by the “invisible” first function argument.

Interrupt Convention

Interrupts are managed on an interrupt stack that is separate from the normal program stack. In the event of an interrupt, the stack pointer is switched to the top of the interrupt stack minus 32 where all the registers are saved according to the convention shown in Table 11.

Table 11: Interrupt Frame Layout

	Register		
Position	**Small**	**Medium**	**Large**
Top of Interrupt Stack – 1	R11	R9	R11
Top of Interrupt Stack – 2	R10	R8	R10
Top of Interrupt Stack – 3	R31	R31	R31
Top of Interrupt Stack – 4	R30	R30	R30
Top of Interrupt Stack – 5	R29	R29	R29
Top of Interrupt Stack – 6	R28	R28	R28
Top of Interrupt Stack – 7	R7	R7	R7
Top of Interrupt Stack – 8	R6	R6	R6
Top of Interrupt Stack – 9	R5	R5	R5
Top of Interrupt Stack – 10	R4	R4	R4
Top of Interrupt Stack – 11	R3	R3	R3
Top of Interrupt Stack – 12	R2	R2	R2
Top of Interrupt Stack – 13	R1	R1	R1
Top of Interrupt Stack – 14	R0	R0	R0

The first four bytes of the scratchpad memory area are reserved to set up the interrupt stack in the event of an interrupt. The compiler will generate code to setup the interrupt stack frame suitable for an interrupt handler in the prologue of the function that has the "interrupt" attribute. For this interrupt handler to link correctly, it must be named "__IRQ". An example is shown in Figure 8.

Figure 8: LatticeMico8 Interrupt Handler

```
__attribute ((interrupt)) __IRQ (void)
{
  // user's interrupt handling code
}
```

Acessing LatticeMico8 Memory Regions

As explained in ""“Memory Regions” on page 5, the LatticeMico8 architecture defines three distinct memory regions - PROM, Scratchpad, and Peripheral (I/O).

Scratchpad

The LatticeMico8 Scratchpad can be read from (or written to) using LatticeMico8 instructions - lsp, lspi, ssp, and sspi - regardless of whether it is internal or external to the microcontroller. The developer should note that the LatticeMico8 compiler always defaults to the Scratchpad for its data reads/writes. That is, all memory accesses are always implemented using these instructions unless otherwise stated.

Note

The size and location of the LatticeMico8 Scratchpad is configurable. The software developer should note that MSB restricts the Scratchpad, regardless of its size, to within 0x00000000 - 0x7FFFFFFF. Any MSB component that falls within this range, as well as, falls within the Scratchpad is accessed using the aforementioned LatticeMico8 instructions.

Peripheral

The LatticeMico8 Peripheral (I/O) region can be read from (or written to) using LatticeMico8 instructions - import, importi, export, and exporti. The developer should note that the LatticeMico8 compiler does not use these instructions for data reads/writes unless explicitly directed to do so. There are two ways in which the developer can instruct the compiler to use these instructions for a particular data access:

1. Inlined Assembly - The developer can access data using inlined assembly that uses these instructions.
2. Builtin Function - The LatticeMico8 compiler provides two "builtin" functions that can be used by the software developer in his code when he needs to access an address within the Peripheral region. The functions are shown in Table 12.

Table 12: Builtin Functions

Function	Effect
void __builtin_export (char value, size_t address)	Generates an export or exporti instruction

Table 12: Builtin Functions

Function	Effect
char __builtin_import (size_t address)	Generated an import or importi instruction. The result of the import instruction is the returned value.

Note: The size of size_t type reflects the size of pointers and is dictated by the memory mode used. Refer to Table 8 for the number of bytes needed for a pointer

Note

The size and location of the LatticeMico8 Peripheral region is configurable. The software developer should note that MSB restricts the Peripheral region, regardless of its size, to within 0x80000000 - 0xFFFFFFFF. Any MSB component that falls within this range, as well as, falls within the Peripheral region is only accessible using the two mechanisms outlined earlier.

PROM

The LatticeMico8 PROM can either be internal to the microcontroller, or can be externally located within a non-volatile memory such as SPI flash. When the PROM is internal to the microcontroller, it cannot be modified via the LatticeMico8 instruction set. The external PROM can only be modified when it falls within the LatticeMico8 Scratchpad.

Index