LatticeMico8 Microcontroller User's Guide

September 2014 **Reference Design RD1026**

Introduction

The LatticeMico8™ is an 8-bit microcontroller optimized for Field Programmable Gate Arrays (FPGAs) and Programmable Logic Device architectures from Lattice. Combining a full 18-bit wide instruction set with 16 or 32 general purpose registers, the LatticeMico8 is a flexible Verilog and VHDL reference design suitable for a wide variety of markets, including communications, consumer, computer, medical, industrial and automotive. The core consumes minimal device resources, less than 200 Look Up Tables (LUTs) in the smallest configuration, while maintaining a broad feature set.

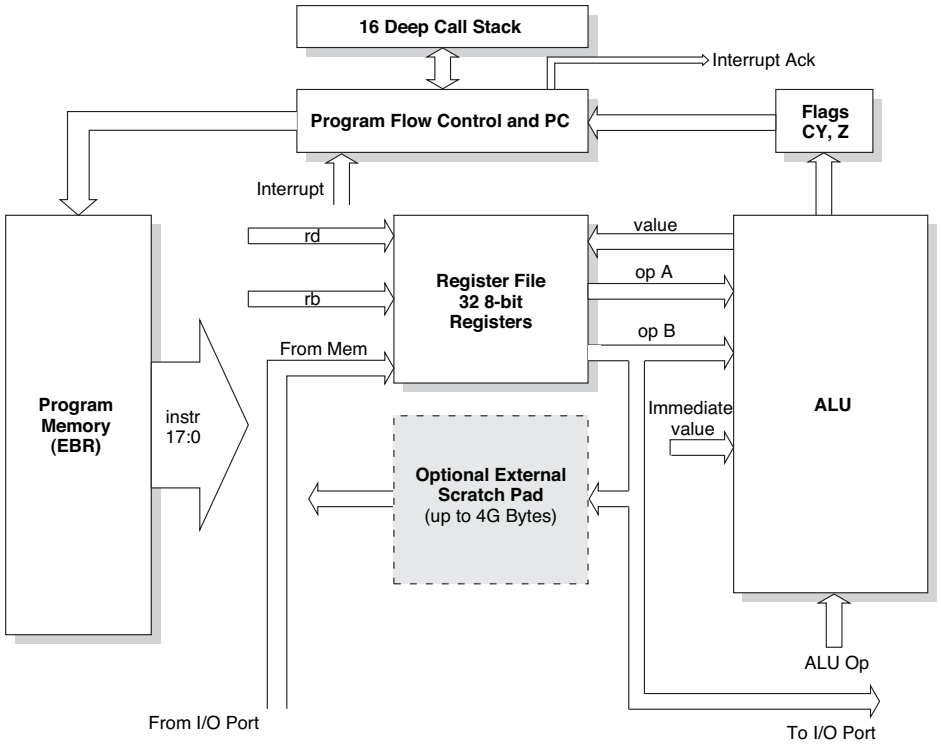
Features

- 8-bit Data Path
- 18-bit Wide Instructions
- Configurable 16 or 32 General Purpose Registers
- Scratch Pad Memory
- Input/Output is Performed Using Paged "Ports" (256 Ports/Page)
- Two/Three Cycles per Instruction
- Lattice UART Reference Design Peripheral

Functional Description

Figure 1 shows the LatticeMico8 microcontroller block diagram.

Figure 1. LatticeMico8 Microcontroller Block Diagram



Exception Vectors

The LatticeMico8 provides two exception vectors. One vector address is used when the processor receives a reset, the other when the processor receives an interrupt.

Address	Function
0	External Int
1	Reset

Address 0 should contain either an iret or unconditional branch instruction.

General Purpose Registers

The LatticeMico8 processor has either 16 or 32 8-bit general purpose registers. The registers are implemented using a dual port distributed memory. The number of registers is configured prior to synthesizing the processor core.

The LatticeMico8 opcode set permits the processor to access 32 registers. When the LatticeMico8 is configured with 16 registers any opcode reference to R16 to R31 maps to R0 to R15 respectively.

Page Pointers

LatticeMico8 can directly access 256 memory locations. In order to increase the amount of memory it can address, LatticeMico8 implements page pointers. General purpose registers R15, R14, and R13 have shadow registers. The external address is the concatenation of R15, R14, R13 and the address generated as a result of a direct/indirect memory opcode, where the output from these shadow registers becomes high-order address bits. This permits the LatticeMico8 to address up to 4 GB of memory using 16M 256-byte pages.

The width of LatticeMico8 address is configurable from eight to 32 bits wide. When LatticeMico8 address bus size is 8 bits wide, the address is generated directly from the opcode being executed. When LatticeMico8 address bus size is from 9 to 16 bits wide, the address bus presents the concatenation of R13 and 8 bits from the opcode being executed. When LatticeMico8 address bus size is from 17 to 32 bits wide, the address bus presents the concatenation of R15, R14, R13 and 8 bits from the opcode being executed. The high-order address bits controlled by R15, R14, and R13 become active when the respective register is updated. The low-order 8 bits of the address bus are valid during the second clock of the instruction and remain valid until the cycle terminates.

Scratch Pad RAM

LatticeMico8 provides an independent memory space that is designed to be used for scratch pad memory. The size of this scratch pad can be configured from 32 bytes to 4G bytes. Page pointers are used when the scratch pad size is larger than 256 bytes.

The scratch pad memory is always external. Direct addressing is used to access the first 32 bytes in the scratch pad regardless of which scratch pad page is active. Indirect addressing can access all 256 bytes of the current active scratch pad page.

Hardware (Circular) Call Stack

When a `call` instruction is executed, the address of the next instruction is pushed onto the call stack, a `ret` (return) instruction will pop the stack and continue execution from the location at the top of the stack.

During an interrupt, the address of the next instruction is pushed onto the call stack. The processor jumps to the interrupt vector at address 00000. Following an `iret` (return from interrupt) instruction the top-most address in the call stack is popped, and execution resumes from the address retrieved from the stack.

The stack is implemented as a 16-entry (default) circular buffer and any program execution will continue from an undefined location in case of a stack overflow or underflow. A synthesis parameter is available to adjust the size of the call stack.

Interrupt Handling

The microcontroller has one interrupt source, which is level-sensitive. The interrupt can be enabled or disabled by software (`cli` = clear interrupt, `sti` = set interrupt). When an interrupt is received, the address of the next instruction is pushed into the call stack and the microcontroller continues execution from the interrupt vector (address 0). The flags (carry and zero) are pushed onto the stack along with the return address. The `interrupt ack` line is set high and the acknowledge line is held high for the entire duration of interrupt handling. Once the interrupt has been acknowledged the interrupt line should be set to 0.

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 and restored. The interrupt acknowledge line is set to low.

The microcontroller cannot handle nested interrupts.

Input/Output

The LatticeMico8 external and scratch pad memory transactions occur synchronously to the LatticeMico8's input clock frequency. The external and scratch pad memories share a single address bus and an output data bus. The input data for each memory is supplied on independent data buses.

The first 32 memory addresses can be accessed using either direct or indirect memory modes. The remaining 224 memory locations can be accessed using only indirect addressing modes.

Figure 2 shows a v.2.4 memory transaction. The address and read/write strobe both appear in the second clock of the processor's decode/execute cycle. The address and strobe only appear for a single clock pulse.

The scratch pad memory read and write strobes, ext_mem_rd/ext_mem_wr, go active as the result of the lsp, lspi, ssp, and sspi opcodes.

The external port strobes, ext_io_rd/ext_io_wr, go active in response to the import, importi, export, exporti opcodes.

The v.3.0 (and later) LatticeMico8 implementations modify the memory access times. Both the scratch pad and external memory cycle times are increased by one clock cycle. Figure 3 shows an example of the v.3.0 bus cycle. The v.3.0 bus cycle presents the address for a minimum of two clock cycles. Write data is also presented for a minimum of two clock cycles. Read data is captured at the rising clock edge of the third clock cycle.

The v.3.0 core also adds an external ready input. When driven LOW at the beginning of the third clock cycle a wait state is inserted. Wait states continue to be asserted until the READY input is driven HIGH coincident to a rising clock edge.

The transition to a three-clock cycle memory transaction in v.3.0 permits use of the FPGA's EBR memory. The Lattice EBR requires address be present for one clock prior to the data being read/written. The v.2.4 memory cycle was incompatible with the EBR required behavior.

Figure 2. Version 2.4 Memory Transaction

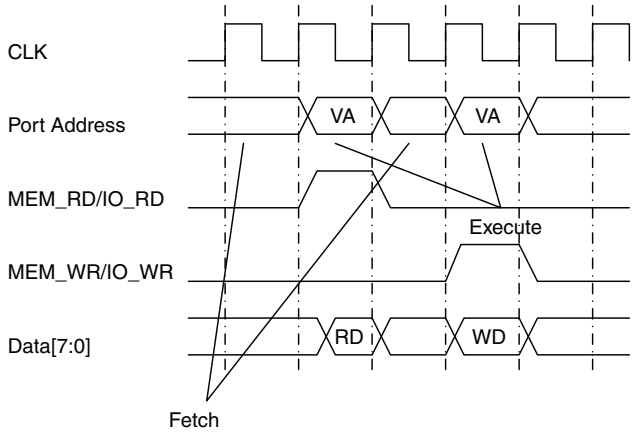
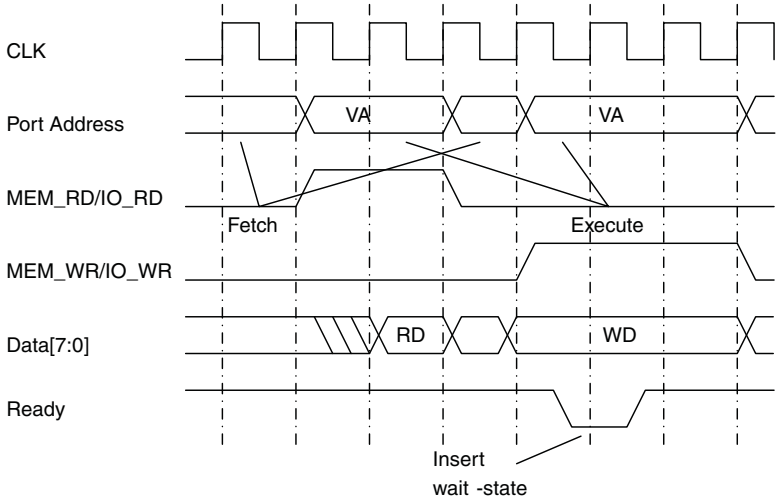


Figure 3. Version 3.0 Bus Cycle Example



LatticeMico8 v.3.15 Enhancements

Version 3.15 of the LatticeMico8 adds some additional capabilities over earlier versions. (Note: Due to improvements in the hardware, any code written for earlier versions of LatticeMico8 is not compatible).

- Increased the number of shadow registers used for extended external addressing. Previously, R14 and R15 had shadow registers. Version 3.15 offers three shadow registers for R13, R14 and R15.
- Increased page pointers from 64K pages to 16M pages.
- Increased the total addressable data memory to 4G Bytes.

LatticeMico8 v.3.1 Enhancements

Version 3.1 of the LatticeMico8 adds some additional capabilities over earlier versions.

- The interrupt handling is fixed. C and Z are correctly pushed to the stack
- This version also fixes asynchronous assertion of RESET
- All the branch instructions now have +/- 2K range
- Opcode decode 0x3c000 is now part of the reserved opcode for future use

LatticeMico8 v.3.0 Enhancements

Version 3.0 of the LatticeMico8 adds some additional capabilities over earlier versions.

- Addition of a READY signal for memory transactions
- 3 clock cycle memory transactions to support EBR and READY
- As described in the previous section the LatticeMico8 memory cycle times have increased by one clock. This gives decode logic time to determine if a memory transaction needs to be lengthened to accommodate slow memory and peripheral devices.

Increase in Instruction PROM Memory from 512 Lines to User-Defined Depth
The instruction memory size is now configurable using a passed HDL parameter. Sizes from 512 to 4096 lines of code have been tested.

Unconditional Branch/Call Instructions Increased to +/- 2K Instruction Range
The increase in instruction store makes having branch and call instructions with a greater range desirable. Unconditional branch and call opcodes can now be created with a +2047/-2048 range.

Family-Specific Modules Implemented Using PMI
The v.2.4 LatticeMico8 source code was written to support the MachXO™ and LatticeXP™ devices. Migrating between Lattice FPGA families was a bit of effort. Version 3.0 permits any Lattice FPGA to be a LatticeMico8 host by simply changing the device selected in ispLEVER®.

Flags Pushed Onto the Call Stack
The C and Z flags are pushed onto the stack following any call or interrupt.

LatticeMico8 Synthesis Parameters

The LatticeMico8 core is reconfigurable. There are many parameters available to allow you to tailor the core to your design needs.

Table 1. LatticeMico8 Synthesis Parameters

Parameter Name	Function
FAMILY_NAME	This is a text entry field that is only used during simulation. The value is used to determine the behavioral model to use for instantiated Parameterized Module Instantiation (PMI) elements. Valid entries for this parameter can be found in the ispLEVER Help. This field is not used during synthesis or place and route. The target FPGA device can be changed in ispLEVER, and a new FPGA bitstream image generated without the need to update the FAMILY_NAME entry. ModelSim® can override the value in the HDL using a command line switch when the HDL is compiled.
PROM_FILE	This is a text entry field that determines the opcode data to be loaded into the LatticeMico8 program memory. This parameter can be explicitly entered in the HDL file, or can be updated as a synthesis parameter from Synplify®, Precision® RTL, or ModelSim.
PORT_AW	This defines the number of low order address bits. The value must be less than or equal to 8. The default value is 8, which permits the LatticeMico8 to address up to 256 external ports.
EXT_AW	This defines the size, in bits, of the external address bus. The parameter must be greater than or equal to PORT_AW. The default value is 8, which permits the LatticeMico8 to address 256 ports.
PROM_AW	This defines the number of address bits assigned to the LatticeMico8 program memory. The default value is 9, which permits up to 512 opcodes to be stored.
PROM_AD	This is the number of opcodes the program memory can store and must always be 2^PROM_AW. The default value is 512 (i.e. 2^9).
REGISTERS_16	This parameter determines how many registers the LatticeMico8 core has. For VHDL the field is a text entry that can be set to TRUE or FALSE. For Verilog the parameter is an integer field that can be either 0 or 1. When the REGISTERS_16 entry is FALSE/0 the LatticeMico8 will have 32 general purpose registers. When it is TRUE/1 it will have 16 general purpose registers.
PGM_STACK_AW	This defines the number of address bits assigned to the LatticeMico8 call stack. The default value is 4, which permits the call stack to hold 16 elements.
PGM_STACK_AD	This defines the depth of the call stack and must always be 2^PGM_STACK_AW. The default value is 16 (i.e. 2^4).

Instruction Set Description and Encoding

Instruction Set Reference Card:

Operation		Action	Flags
add	Rd, Rb	Rd = Rd + Rb	CZ
addc	Rd, Rb	Rd = Rd + Rb + Carry	CZ
addi	Rd, C	Rd = Rd + Const	CZ
addic	Rd, Rb	Rd = Rd + Const + Carry	CZ
and	Rd, Rb	Rd = Rd & Rb	Z
andi	Rd, C	Rd = Rd & Const	Z
b	label	Branch unconditionally	--
bc	label	Branch on carry flag = 1	--
bnc	label	Branch on carry flag = 0	--
bnz	label	Branch on zero flag = 0	--
bz	label	Branch on zero flag = 1	--
call	label	Call function	--
callc	label	Call function on carry = 1	--
callnc	label	Call function on carry = 0	--
callnz	label	Call function on zero = 0	--
callz	label	Call function on zero = 1	--
clrc		Carry flag = 0	C
clri		Disable interrupts	--
clrz		Zero flag = 0	Z
cmp	Rd, Rb	Rd - Rb	CZ
cmpi	Rd, C	Rd - Const	CZ
export	Rd, port#	(Port#) = Rd	--
exporti	Rd, Rb	(Rb), Rd	--
import	Rd, port#	Rd = (Port#)	--
importi	Rd, Rb	Rd = (Rb)	--
iret		Return from interrupt	--
lsp	Rd, sp#	Rd = (sp#)	--
lspi	Rd, Rb	Rd = (Rb)	--
mov	Rd, Rb	Rd = Rb	--
movi	Rd, C	Rd = Const	--
or	Rd, Rb	Rd = Rd | Rb	Z
ori	Rd, C	Rd = Rd | Const	Z
ret		Return from Call	--
rol	Rd, Rb	Rd = Rb << 1, Rb(0) = Rb(7)	Z
rolc	Rd, Rb	Rd = C:Rb << 1, Rb(0) = C	CZ
ror	Rd, Rb	Rd = Rb >> 1, Rb(7) = Rb(0)	Z
rorc	Rd, Rb	Rd = C:Rb >> 1, C = Rb(0)	CZ
setc		Carry flag = 1	C
seti		Enable intterupts	--
setz		Zero flag = 1	Z
ssp	Rd, sp#	(sp#) = Rd	--
sspi	Rd, Rb	(Rb) = Rd	--
sub	Rd, Rb	Rd = Rd - Rb	CZ
subc	Rd, Rb	Rd = Rd - Rb - Carry	CZ
subi	Rd, C	Rd = Rd - Const	CZ
subic	Rd, C	Rd = Rd - Const - Carry	CZ
test	Rd, Rb	Rd & Rb	Z
testi	Rd, C	Rd & Const	Z
xor	Rd, Rb	Rd = Rd ^ Rb	Z
xori	Rd, C	Rd = Rd ^ Const	Z

Please note that for all Branch and Call instructions, the signed offset is represented as binary 2's complement.

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.

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.

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.

ADDIC Rd, CC

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.

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.

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.

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.

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.

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	Yes

Rd = Rb (move register to register)

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	Yes

Rd = CCCCCCCC (move constant into register)

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.

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.

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, CC

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.

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.

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, CC

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

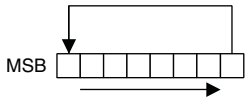
Perform a bitwise AND between Rd and Constant, update the zero flag. The result of the AND operation is not written back.

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

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

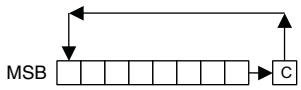


Rotate right. Register Rb is shifted right one bit, the highest order bit is replaced with the lowest order bit. The result is written back to Register Rd. The zero flag is set to 1 if all the bits of the result are 0.

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

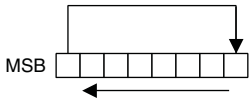


Rotate right through carry. The contents of Register Rb are shifted right one bit, the carry flag is shifted into the highest order bit, the lowest order bit is shifted into the carry flag. The result is written back to Register Rd. The zero flag is set to 1 if all the bits of the result are 0.

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

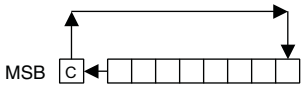


Rotate left. Register Rb is shifted left by one bit. The highest order bit is shifted into the lowest order bit. The result is written back to Register Rd. The zero flag is set to 1 if all the bits of the result are 0.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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
Yes	Yes

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

Unconditional return. PC is set to the value on the top of the call stack. The CY and Z flags are restored from the call stack.

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.

IMPORT Rd, Port#
INP 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 = Value from Port (Port#)
Read value from port number (Port#) and write into register Rd. Port # can be 0-31.

IMPORTI Rd, Rb
INPI 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 = Value from Port # in Register Rb
Indirect read of port. Value is read from port number in register Rb. Port number can be 0-255 of active page.

EXPORT Rd, Port#
OUTP 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

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

EXPORTI Rd, Rb
OUTPI 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
Output value of Register Rd to Port# designated by Register Rb. Port# can be 0-255 of active page.

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 (Rb)

Load from scratch pad memory indirect. Load the value from the scratch pad location designated by Register Rb into Register Rd. The location address can be 0-255 of active page.

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 (Rb) = Rd

Store into scratch pad memory indirect. Store value of Register Rd, into scratch pad memory location designated by Register Rb. The location address can be 0-255 of active page.

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.

Assembler and Instruction Set Simulator

The software tools for the LatticeMico8 microcontroller include an Assembler and an Instruction Set Simulator, both developed in C. The purpose of the Assembler is to generate an Embedded Block RAM (EBR) initialization file from a text assembler input file. The purpose of the Simulator is to execute a program in the host environment. This section describes the use of these tools.

Assembler

The Assembler reads in a text assembler source file (default extension .s) and creates one of the following as output:

- Hexadecimal output file (can be used by Module Manager)
- Binary output file (can be used by Module Manager)
- Memory output file (can be used by Module Manager)
- Verilog initialization file (included in design before synthesis)

In addition to these outputs, the Assembler can also generate an Assembler listing file.

Command Line
isp8asm -option1 -option2 ... <input filename>

Command Line Options

Option	Comment
-o <filename>	Fully qualified name of the output file.
-s <Program Rom Size>	Default 512 bytes
-l <filename>	Fully qualified name of the listing file.
-vx	Generate output in 18-bit hexadecimal (default)
-vb	Generate output in binary

-ve	Generate output in Verilog "INIT" format
-vm	Generate output in MEM file format
-vr	Generate output in 8-bit hexadecimal
-?	Help message

Instructions
The Assembler supports all instructions as described in the Instruction Set section.

Pseudo-Ops
The Assembler supports the following pseudo-ops:

Option	Comment
nop	Expanded by the Assembler to mov R0,R0. An instruction without side effects.

Labels
Label definitions are any character sequences ending in a ':'. No other instruction or Assembler directives are allowed in the same line as a label definition.

The Assembler allows both forward and backward references to a label (i.e. it is legal to reference a label before it is defined). Both references in the following example are valid. Labels are case-sensitive.

```
BackLabel:
            ...
            ...
            b   BackLabel
            ...
            ...
            b   ForwardLabel
            ...
            ...
ForwardLabel:
```

Comments
The character ';' is used as the start of a comment. Everything following the comment character until a new line is ignored by the Assembler.

Constants
The Assembler accepts constants in various formats.

- **Hexadecimal values**: Hexadecimal constants must be prefixed with "0x" or "0X". (e.g. 0xFF, 0x12, and 0XAB are all valid hexadecimal constants).
- **Octal values**: Octal values must be prefixed with the numeric character '0'. (e.g. 077, 066, and 012 are valid octal constants).
- **Character constants**: Single character constants must be enclosed in single quotation marks. (e.g. 'A', 'v', '9' are all valid character constants).
- **Decimal constants**: Any sequence of decimal numbers can be a valid constant. (e.g. 123, 255, 231 are valid decimal constants).
- **Location counter**: The special character $ (dollar sign) is used to give the current value of the location counter.

Note: The hexadecimal, octal, and decimal constants can be optionally prefixed with a '+' or '-' sign.

Assembler Directives
In addition to the instructions described in the Instruction Set section, the Assembler also supports the following directives. An Assembler directive must be prefixed with a '.' character.

- **.org**: This directive allows code to be placed at specific addresses. The syntax for this directive is:

  .org <constant>

  The constant can be of any form described in the previous section. The Assembler will terminate with an error, if the .org directive is given a location which is less than the current "local counter" value.

- **.equ**: This directive can be used to assign symbolic names to constants. The syntax of the directive is:

```
.equ  <symbolic name>,<constant>
.equ newline,'\n'
...
movi  r2,newline
```

- **.data**: This directive can be used to embed arbitrary data in the Assembler. The syntax for this directive is:

```
.data <constant>
.data "<char><char><char>"
.data 0x0d
.data "Hello World"
```

- **.ebr**: The .ebr directive duplicates either the -vx or the -vr command line switch. Only a single .ebr directive can be inserted into an assembly source file. This directive can be used to target the Assembler output for code space (18 bits) or data space (8 bits). The syntax for this directive is:

```
.ebr 8 ; data type 8-bit hex output
.ebr 18 ; code type 18-bit hex output
```

- **.prom**: This directive sets the maximum number of opcodes the Assembler can generate without issuing an error. The directive duplicates the "-s" command line switch. Only a single .prom directive can be applied to an assembly source file. The syntax for this directive is:

```
.prom 512
```

Figure 4 is an example of the listing generated by the Assembler:

Figure 4. Example of Assembler Generated Listing

```
Loc          Opcode       Opcode
Counter      (Hex)        (Bin)
0x0000       0x33001      110011000000000001      b           start
0x0001             start:
0x0001       0x10000      010000000000000000      nop
0x0002             add:
0x0002       0x12055      010010000001010101      movi        R00,0x55
0x0003       0x12105      010010000100000101      movi        R01,0x05
0x0004       0x12203      010010001000000011      movi        R02,0x03
0x0005       0x08110      001000000100010000      add         R01,R02
0x0006       0x0A101      001010000100000001      addi        R01,0x01
0x0007       0x10308      010000001100001000      mov         R03,R01
0x0008       0x10410      010000010000010000      mov         R04,R02
0x0009       0x12535      010010010100110101      movi        R05,0x35
0x000A       0x12643      010010011001000011      movi        R06,0x43
0x000B       0x08628      001000011000101000      add         R06,R05
0x000C       0x0A613      001010011000010011      addi        R06,0x13
0x000D       0x10728      010000011100101000      mov         R07,R05
                                    .
                                    .
                                    .
```

Building Assembler from Source

Although Lattice provides precompiled binary files, the source is available for compilation. The following commands should be used in the Unix and Windows environments.

- **Unix and Cygwin Environments:**
  gcc -o isp8asm isp8asm.c
- **Windows Environment:**
  cl -o isp8asm_win isp8asm.c

Instruction Set Simulator

The software tools for LatticeMico8 include an Instruction Set Simulator for the microcontroller which allows programs developed for the microcontroller to be run and debugged on a host platform. The Simulator can also be used to generate a disassembly listing of a LatticeMico8 program. The Simulator takes as input the memory output file of the Assembler. It emulates the instruction execution of the LatticeMico8 in software. Please note that the Simulator does not handle interrupts.

Command Line

<executable filename> -option1 -option2 ... <prom filename> <scratch pad filename>

Command Line Options

Option	Comment
-p <Program Rom Size>	Default is 512 opcodes.
-ix	Program file is in hexadecimal format (default). This is the file generated by the Assembler with the -vx options (default).
-ib	Program file is in binary format. This is the file generated by the Assembler with the -vb option.

-t	Trace the execution of the program. The Simulator will generate a trace as it executes each instruction. It will also print the modified value of any register (if the instruction modifies a register value).
-d	Generate a disassembly of the program specified by the PROM file.

Simulator Interactions

The **import**, **importi** and **export**, **exporti** instructions can be used to interact with the simulator. When an **export**, **exporti** instruction is executed, the simulator will print the value of the port number as well as the contents of the exported register. If the port number is 0xFF, the simulator will terminate with an exit code identical to the value of the exported register. When an **import**, **importi** instruction is executed, the simulator will issue a prompt containing the port number and read in values from the standard input (stdin). The following figure shows an example of a traced simulation. The v.3.0 simulator only implements an 8-bit external address bus.

Figure 5. Example of Trace Simulation

```
0x00001     0x10000         mov             R00,R00
0x00002     0x12055         movi            R00,0x55
            R00 = 0x55
0x00003     0x12105         movi            R01,0x05
            R01 = 0x05
0x00004     0x12203         movi            R02,0x03
            R02 = 0x03
0x00005     0x08110         add             R01,R02
            R01 = 0x08
0x00006     0x0A101         addi            R01,0x01
            R01 = 0x09
0x00007     0x10308         mov             R03,R01
            R03 = 0x09
0x00008     0x10410         mov             R04,R02
            R04 = 0x03
0x00009     0x12535         movi            R05,0x35
            R05 = 0x35
0x0000A     0x12643         movi            R06,0x43
            R06 = 0x43
0x0000B     0x08628         add             R06,R05
            R06 = 0x78
0x0000C     0x0A613         addi            R06,0x13
            R06 = 0x8B
0x0000D     0x10728         mov             R07,R05
            R07 = 0x35
0x0000E     0x10830         mov             R08,R06
            R08 = 0x8B
0x0000F     0x12916         movi            R09,0x16
            R09 = 0x16
0x00010     0x12ADF         movi            R10,0xDF
            R10 = 0xDF
                  .
                  .
                  .
```

Building Simulator from Source

Although Lattice provides precompiled binary files, the source is available for compilation. The following commands should be used in the Unix and Windows environments.

- **Unix and Cygwin Environments:**
  gcc -o isp8sim isp8sim.c
- **Windows Environment:**
  cl -o isp8sim_win isp8sim.c

Example

To display the features and capabilities of the LatticeMico8, a demonstration example is also available. It demonstrates the interaction between the timer and the controller and the interrupt capability.

```
# This program will allow user to run a fibonacci number
# generator and updown counter. This program responds to
# the interrupt from the user (through Orcastra).
# When there is an interrupt, the program will halt the current program,
# and execute the int_handler function. When the intr_handler function
# is done, the program will continue from its last position

      b      int_handler
      nop
      nop
      seti                    # set the program to be able to receive interrupt
      nop
      nop
      b  start

start:

      import r5, 5


      mov r6, r5
      andi r5, 0xf0           # masking r5 to decide type of program
      mov r7, r5

      mov r5, r6
      andi r5, 0x0f           # masking r5 to get the speed
      mov r25, r5

      cmpi r7, 0x10
      bz phase2
      cmpi r7, 0x20
      bz phase2
      b start

phase2:
      cmpi r25, 0x01
      bz phase3
      cmpi r25, 0x02
      bz phase3
      cmpi r25, 0x03
      bz phase3
```

```
      cmpi r25, 0x04
      bz phase3
      b start

phase3:

      cmpi r7, 0x10
      bz fibo
      cmpi r7, 0x20           # 1 = fibonacci, 2 = counter
      bz counter
      b start
```

Implementation

Table 2. Performance and Resource Utilization

Config. Number	Description	Device Family	Language	LUTs	Registers	SLICEs	f_{MAX}(MHz)
1	16 registers, 32-byte Ext. SP, 512 PROM, 8-bit Ext. Address	ECP5U[1]	Verilog-LSE	268	72	162	78.04
			Verilog-Syn	294	65	170	78.42
		ECP5UM[2]	Verilog-LSE	268	72	162	68.51
			Verilog-Syn	294	65	170	71
		LatticeEC™[3]		264	65	157	71.69
		LatticeECP™[4]		264	65	157	71.69
		LatticeECP2™[5]		285	65	166	87.6
		LatticeECP2M™[6]		285	65	166	82.25
		LatticeECP3™[7]		307	67	177	92.21
		LatticeSC™[8]		261	82	153	106.04
		LatticeSCM™[9]		261	82	153	106.04
		LatticeXP™[10]		264	65	157	63.66
		LatticeXP2™[11]		289	65	171	77.86
		MachXO™[12]		250	64	125	65.79
		MachXO2™[13]	Verilog-LSE	263	72	133	54.40
			Verilog-Syn	265	64	133	53.34
		MachXO3L™[14]	Verilog-LSE	263	72	133	66.72
			Verilog-Syn	265	64	134	54.42

Config. Number	Description	Device Family	Language	LUTs	Registers	SLICEs	f_{MAX}(MHz)
2	32 registers, 32-byte Ext. SP, 512 PROM, 8-bit Ext. Address	ECP5U[1]	Verilog-LSE	311	72	184	63.65
			Verilog-Syn	328	65	182	67.08
		ECP5UM[2]	Verilog-LSE	310	72	184	66.46
			Verilog-Syn	328	65	182	66.92
		LatticeEC[3]		318	65	183	63.8
		LatticeECP[4]		318	65	183	63.8
		LatticeECP2[5]		323	65	184	60.47
		LatticeECP2M[6]		323	65	184	73.15
		LatticeECP3[7]		364	65	201	89.08
		LatticeSC[8]		322	81	182	92.76
		LatticeSCM[9]		322	81	182	92.76
		LatticeXP[10]		318	65	183	56.12
		LatticeXP2[11]		326	65	187	73.55
		MachXO[12]		297	64	149	56.25
		MachXO2[13]	Verilog-LSE	310	72	157	54.38
			Verilog-Syn	303	64	154	48.58
		MachXO3L[14]	Verilog-LSE	310	72	157	57.82
			Verilog-Syn	303	64	154	51.27
3	16 registers, 32-byte ext. SP, 512 PROM, 16-bit ext. address	ECP5U[1]	Verilog-LSE	260	78	159	73.36
			Verilog-Syn	302	74	172	76.59
		ECP5UM[2]	Verilog-LSE	260	78	159	73.28
			Verilog-Syn	302	74	172	74.59
		LatticeEC[3]		279	73	161	71.47
		LatticeECP[4]		279	73	161	71.47
		LatticeECP2[5]		305	73	177	71.76
		LatticeECP2M[6]		305	73	177	79.89
		LatticeECP3[7]		305	74	170	92.27
		LatticeSC[8]		270	90	158	104.53
		LatticeSCM[9]		270	90	158	104.53
		LatticeXP[10]		279	73	161	56.54
		LatticeXP2[11]		303	73	176	77.94
		MachXO[12]		259	72	130	62.63
		MachXO2[13]	Verilog-LSE	267	80	134	57.03
			Verilog-Syn	273	72	139	52.20
		MachXO3L[14]	Verilog-LSE	267	80	134	62.18
			Verilog-Syn	273	72	139	60.96

Config. Number	Description	Device Family	Language	LUTs	Registers	SLICEs	f_{MAX}(MHz)
4	32 registers, 32-byte ext. SP, 512 PROM, 16-bit ext. address	ECP5U[1]	Verilog-LSE	303	78	180	63.12
			Verilog-Syn	338	71	189	54.25
		ECP5UM[2]	Verilog-LSE	303	78	180	67.76
			Verilog-Syn	338	71	189	64.87
		LatticeEC[3]		337	71	188	65.13
		LatticeECP[4]		337	71	188	65.13
		LatticeECP2[5]		342	71	195	79.88
		LatticeECP2M[6]		342	71	195	75.25
		LatticeECP3[7]		355	73	202	84.5
		LatticeSC[8]		320	90	178	91.69
		LatticeSCM[9]		320	90	178	91.69
		LatticeXP[10]		337	71	188	61.04
		LatticeXP2[11]		352	71	200	72.01
		MachXO[12]		250	64	125	65.79
		MachXO2[13]	Verilog-LSE	310	80	156	50.72
			Verilog-Syn	310	70	157	51.27
		MachXO3L[14]	Verilog-LSE	310	80	156	56.36
			Verilog-Syn	310	70	157	54.59

1. Performance and utilization characteristics are generated using LFE5U-85F-6MG285CES with Lattice Diamond® 3.3 design software.
2. Performance and utilization characteristics are generated using LFE5UM-45F-6MG285C with Lattice Diamond 3.3 design software.
3. Performance and utilization characteristics are generated using LFEC15E-5F484C with Lattice Diamond 3.3 design software.
4. Performance and utilization characteristics are generated using LFECP15E-5F484C with Lattice Diamond 3.3 design software.
5. Performance and utilization characteristics are generated using LFE2-50E-5F484C with Lattice Diamond 3.3 design software.
6. Performance and utilization characteristics are generated using LFE2M35E-5F484C with Lattice Diamond 3.3 design software.
7. Performance and utilization characteristics are generated using LFE3-35EA-8FN484C with Lattice Diamond 3.3 design software.
8. Performance and utilization characteristics are generated using LFSC3GA80E-6FC1152C with Lattice Diamond 3.3 design software.
9. Performance and utilization characteristics are generated using LFSCM3GA80EP1-6FC1152C with Lattice Diamond 3.3 design software.
10. Performance and utilization characteristics are generated using LFXP20E-5F256C with Lattice Diamond 3.3 design software.
11. Performance and utilization characteristics are generated using LFXP2-8E-6TN144C with Lattice Diamond 3.3 design software.
12. Performance and utilization characteristics are generated using LCMXO1200C-4T100C with Lattice Diamond 3.3 design software.
13. Performance and utilization characteristics are generated using LCMXO2-1200HC-5MG132C with Lattice Diamond 3.3 design software.
14. Performance and utilization characteristics are generated using LCMXO3L-4300C-6BG256C with Lattice Diamond 3.3 design software.

Technical Support Assistance

e-mail: techsupport@latticesemi.com

Internet: www.latticesemi.com

Revision History

Date	Version	Change Summary
—	—	Previous Lattice releases.
March 2007	01.3	Updated B Label, CALL Label, RET and IRET instruction sets.
April 2007	01.4	Corrected IRET instruction.
February 2008	01.5	Added version 3.0 information.
October 2009	01.6	Extended branch and call range to +2047/-2048 for all types.
June 2010	01.7	Added version 3.1 information.
		Added LatticeXP2 FPGA support.
October 2010	01.8	Added LatticeMico8 version 3.15 information.
November 2010	01.9	Added support for MachXO2 device family.
February 2014	02.0	Updated Page Pointers text section.
		Updated LatticeMico8 v.3.1.5 Enhancements text section.
		Added support for MachXO3L device family.
		Updated Technical Support Assistance information.
		Updated corporate logo.
September 2014	02.1	Updated Table 2, Performance and Resources Utilization. Added Support for Lattice Diamond 3.3 design software. Added LSE support for LatticeXO2. Added support for Lattice ECP5, and Lattice ECP3 device family.