RISC-V MC CPU IP Core – Lattice Propel Builder 2024.1

User Guide

FPGA-IPUG-02252-1.0

May 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Inclusive Language

This document was created consistent with Lattice Semiconductor’s inclusive language policy. In some cases, the language in underlying tools and other items may not yet have been updated. Please refer to Lattice’s inclusive language FAQ 6878 for a cross reference of terms. Note in some cases such as register names and state names it has been necessary to continue to utilize older terminology for compatibility.

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
AHB-L	Advanced High-performance Bus – Lite
CF	Custom Function
CFU	Custom Function Unit
CFU-LI	Custom Function Unit Logic Interface
CPU	Central Processing Unit
CSR	Control and Status Register
DMIPS	Dhrystone MIPS (Million Instructions per Second)
FPGA	Field Programmable Gate Array
GDB	Gnu Debugger
HDL	Hardware Description Language
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group
LUT	Lookup-Table
MC	Micro-Controller (RISC-V for Micro-Controller applications)
OpenOCD	Open On-Chip Debugger
PIC	Programmable Interrupt Controller
RISC-V	Reduced instruction set computer-V (five)
RV32IMC	RISC-V Integer, M and Compressed Instruction Sets
WFI	Wait For Interrupt
CFU-LI	Custom Function Unit Logic Interface

1. Introduction

The Lattice Semiconductor RISC-V MC CPU soft IP contains a 32-bit RISC-V processor core and optional submodules – Timer and Programmable Interrupt Controller (PIC). The CPU core is with instruction and data caches. The CPU core supports RV32IMC instruction set, external interrupts, and debug feature that is JTAG – IEEE 1149.1 compliant. The Timer submodule is a 64-bit real-time counter, which compares a real-time register with another register to assert the timer interrupt. The PIC submodule aggregates up to eight external interrupt inputs into one external interrupt. The submodule registers are accessed by the processor core using a 32-bit Advanced High-performance Bus – Lite (AHB-L) interface.

The design is implemented using Verilog HDL, and it can be configured and generated using the Lattice Propel™ Builder software. It supports ECP5™, ECP5-5G™, Lattice Avant™, MachXO5™-NX, CrossLink™-NX, Certus™-NX, CertusPro™-NX, MachXO3D™, MachXO3™, and MachXO2™ FPGA devices.

1.1. Quick Facts

Table 1.1 presents a summary of the RISC-V MC CPU IP Core.

Table 1.1 RISC-V MC CPU IP Core Quick Facts

IP Requirements	Supported FPGA Family	ECP5, ECP5-5G, Lattice Avant, MachXO5-NX, CrossLink-NX, Certus-NX, CertusPro-NX, MachXO3D, MachXO3L™, MachXO3LF™, MachXO2
Resource Utilization	Targeted Devices	LAE5U, LAE5UM, LFE5U, LFE5UM, LFE5UM5G, LAV-AT, LFMXO5, LIFCL, LFD2NX, LFCPNX, LAMXO3D, LCMXO3D, LCMXO3L, LAMXO3LF, LCMXO3LF, LCMXO2
	Supported User Interfaces	AHB – Lite Interface
	Resources	See Table A.1, Table A.2, and Table A.3.
Design Tool Support	Lattice Implementation	IP Core Version 2.6.0 – Lattice Propel Builder 2024.1, Lattice Radiant™ 2024.1
	Simulation	For a list of supported simulators, see the Lattice Radiant and Lattice Diamond™ software user guide.

1.2. Features

The RISC-V MC soft IP has the following features:

- RV32IMC instruction set
- Five stage pipeline
- Support the AHB-L bus standard for instruction/data ports
- Optional caches including a 4 KB two-way instruction cache and a 4 KB two-way data cache, for Lattice Avant, MachXO5-NX, Certus-NX, CertusPro-NX, and CrossLink-NX devices only
- Optional debug using Gnu Debugger (GDB) and Open On-Chip Debugger (OpenOCD)
- Optional PIC module
- Optional Timer module
- Interrupt and exception handling under Machine Mode
- > 0.7 DMIPS/MHz performance
- > 100 MHz on CrossLink-NX devices, tested on the Hello World template provided by Lattice Propel design environment.
- Custom Function Unit

1.3. Conventions

The nomenclature used in this document is based on Verilog HDL.

1.4. Licensing and Ordering Information

The MC CPU IP is provided at no additional cost with the Lattice Propel design environment. The IP can be fully evaluated in hardware without requiring an IP license string.

2. Functional Descriptions

2.1. Overview

The RISC-V MC CPU IP processes data and instructions while monitoring external interrupts. As shown in Figure 2.1, the CPU IP has a 32-bit processor core and optional submodules. It uses a read-only AHB-L interface for instruction fetch and another AHB-L interface with read/write access for data access. See Table 2.5 and Table 2.6 for the AHB-L Instruction Fetch and Data Accessing ports definition. The CPU core, PIC, Timer, and AHB-L multiplexor run in the system clock domain. The Core Debug runs in both system clock domain and JTAG clock domain.

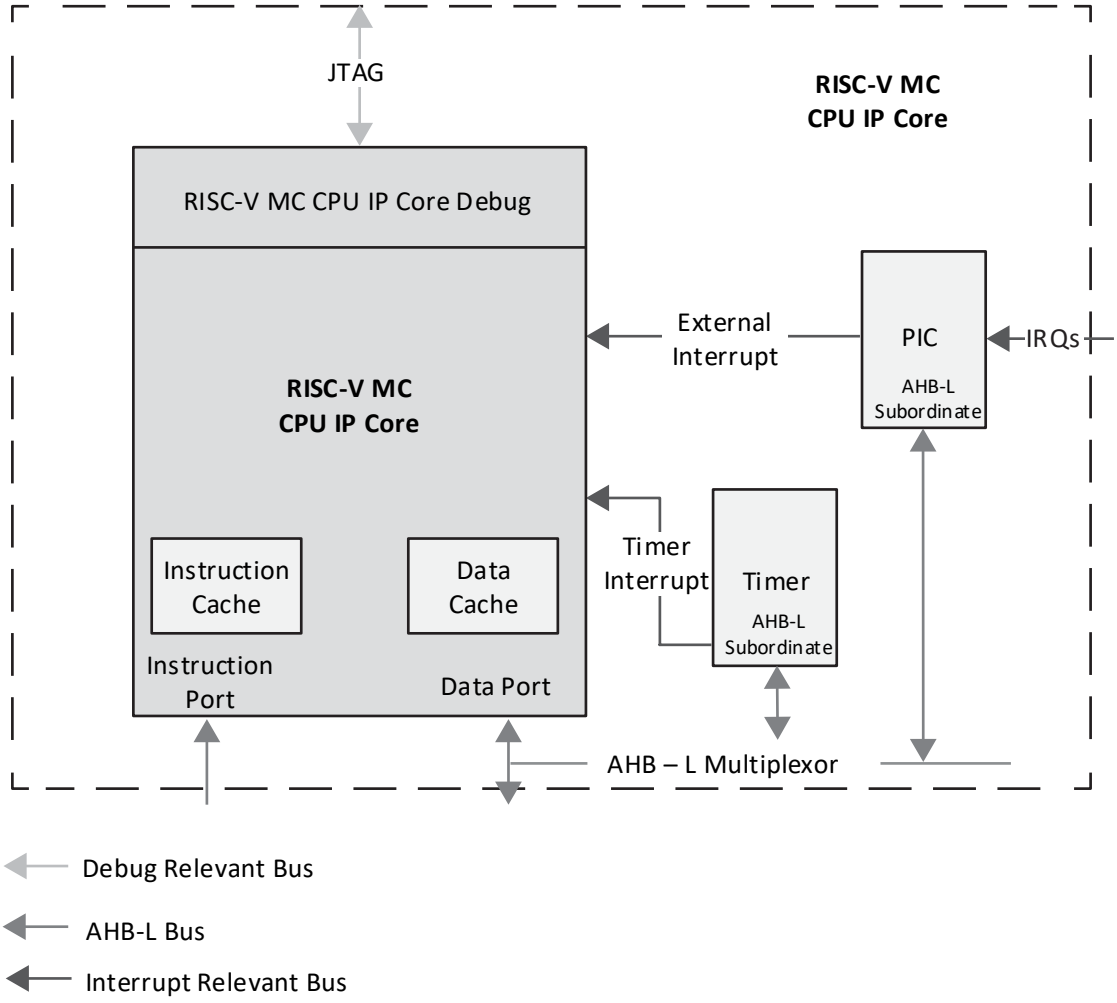

Figure 2.1. RISC-V MC Soft IP Diagram

2.2. Modules Description

2.2.1. RISC-V Processor Core

The processor core follows the RV32IMC instruction set and the M and C extensions are optional. Figure 2.2 shows the processor core block diagram.

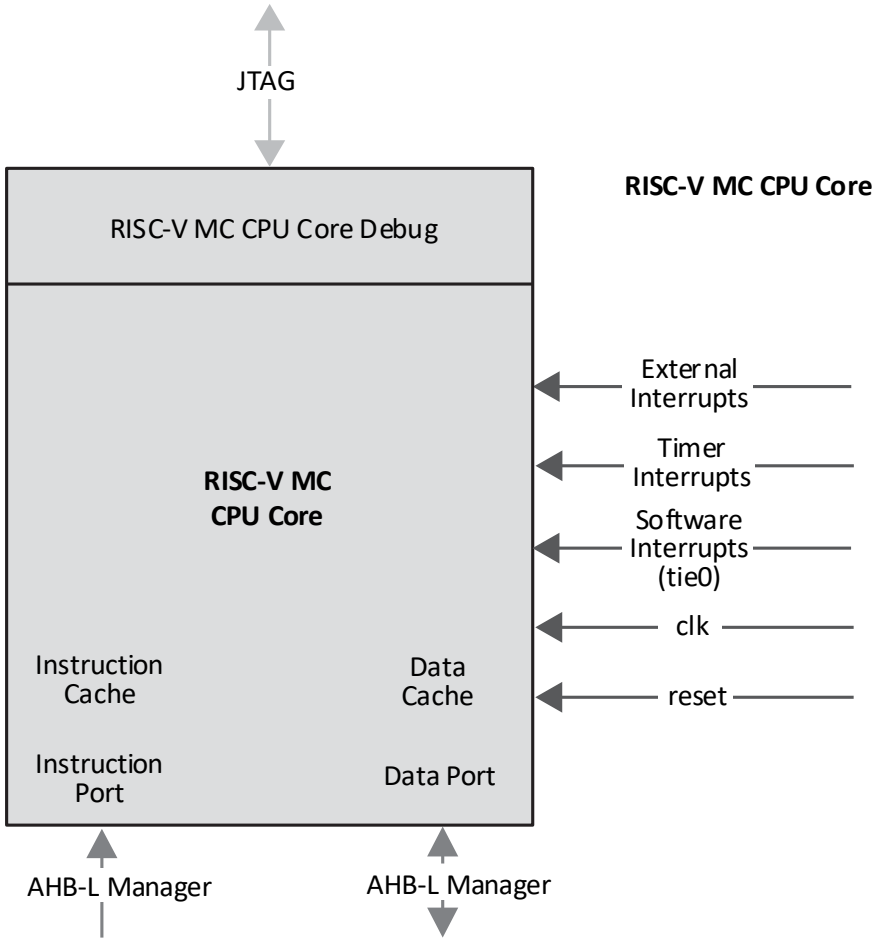

Figure 2.2. RISC-V MC Processor Core Block Diagram

2.2.1.1. Interrupt

The core CPU's interrupts are level sensitive and high active. A given interrupt should remain asserted until cleared by the corresponding interrupt service routine.

2.2.1.2. Exception

If an exception occurs, the processor core stops the corresponding instruction, flushes all previous instructions, and waits until the terminated instruction reaches the writeback stage before jumping to the exception service routine.

2.2.1.3. Low Power Mode

The processor core enters into low power mode with Wait For Interrupt (WFI) instruction. The program counter halts during low power mode, and the processor wakes up if there is an external/timer interrupt.

2.2.1.4. Debug

The processor core supports the IEEE-1149.1 JTAG debug logic with two hardware breakpoints.

To use the debug module, it is required to allow writes from data port to instruction memory in the SoC. Single-port instruction memory is not allowed to debug.

The soft JTAG is available on Lattice Avant, MachXO5-NX, CrossLink-NX, CertusPro-NX, and Certus-NX devices. For more information, refer to Appendix B.

2.2.1.5. Instruction and Data Caches

The processor core supports optional instruction and data caches.

The instruction and data caches are both 4 KB two-way set associative, each cache line contains 32 bytes. The cache strategy for data cache is write through, and the cache eviction policy of both caches is round robin.

The instruction and data caches can be enabled/disabled together by checking/unchecking the CACHE_ENABLE option when instantiating the CPU in Lattice Propel design environment, and the cacheable address range can be configured by setting the CACHEABLE_ADDR_LOW and CACHEABLE_ADDR_HIGH parameters.

When cache is enabled, it is required to store the instructions into the instruction cache range.

To flush the caches, refer to annotations of cache.h in the driver codes. The cache invalidates the corresponding cache line and reloads it from memory the next time it is accessed. Those instructions are accepted only if the cache is enable. If the cache is not enabled, those instructions raise an illegal instruction exception.

It should be noted that the instruction and data caches should only be enabled for Lattice Avant, MachXO5-NX, CrossLink-NX, CertusPro-NX, and Certus-NX devices, as they have enough resources to support the caches.

2.2.1.6. Reset Vector

The reset vector of the processor is 0x0000_0000 and it is fixed.

2.2.1.7. Branch Prediction

Processors with caches use dynamic target prediction for branches and processors without caches do not implement branch prediction.

2.2.1.8. Control and Status Registers

The processor core supports the Control and Status Registers (CSRs) listed in Table 2.1.

Table 2.1. RISC-V Processor Core Control and Status Registers

CSR No.	CSR Name	Access	Fields
0x300	Machine Status (mstatus)	read/write	bit[12:11]: mpp, privilege mode before entering a trap, should always be 2'b11 in machine mode in this CPU core. bit[7]: mpie, mie before entering a trap, updates to mie value when entering a trap. bit[3]: mie, global interrupt enable.
0x301	Machine ISA (misa)	read-only	bit[31:30]: base, hardwired 0x1, stands for RV32. bit[25:0]: extension, stands for the supported ISAs.
0x304	Machine Interrupt Enable (mie)	read/write	bit[11]: meie, machine mode external interrupt enable. bit[7]: mtie, machine mode timer interrupt enable. bit[3]: msie, machine mode software interrupt enable.
0x305	Machine Trap-Vector Base-Address (mtvec)	read/write initialized to 0x20	bit[31:2]: trap vector base address, 4-byte aligned. bit[0]: trap vector mode, all traps set the program counter to the base address in the RISC-V MC CPU core. Bit[1] is not supported. Only 1'0 – direct mode and 1'b1 – vectored mode are available.
0x340	Machine Scratch (mscratch)	read/write	bit[31:0]: in machine mode, it is used to hold a pointer to a machine-mode hart-local context space and is swapped with a user register upon entry to a machine mode trap handler.
0x341	Machine Exception Program Counter (mepc)	read/write	bit[31:0]: when a trap is taken into machine mode, mepc is used to store the address of the instruction that encounters the exception.

CSR No.	CSR Name	Access	Fields
0x342	Machine Cause (mcause)	read-only	bit[31]: 1'b1 – interrupt, 1'b0 – exception bit[3:0]: exception code for interrupt: • 3 – machine software interrupt • 7 – machine timer interrupt • 11 – machine external interrupt for exception : • 0 – instruction address misaligned • 1 – instruction access fault • 2 – illegal instruction • 4 – load address misaligned • 5 – load access fault
0x343	Machine Trap Value (mtval)	read-only	bit[31:0]: When a hardware breakpoint is triggered, or an instruction fetch, load, or store address is misaligned, or an access exception occurs, mtval is written with the fault address. It may also be written with an illegal instruction when an illegal instruction occurs.
0x344	Machine Interrupt Pending (mip)	read/write	bit[11]: meip, machine mode external interrupt pending, read-only. bit[7] mtip, machine mode timer interrupt pending, read-only. bit[3] msip, machine mode software interrupt pending, readable and writable.
0xB00	Machine Cycle (mcycle)	read/write	bit[31:0]: Machine cycle counter
0xB02	Machine Instructions-Retired (minstret)	read/write	bit[31:0]: Machine instructions-retired counter
0xB80	Upper 32 bits of Machine Cycle (mcycleh)	read/write	bit[31:0]: Upper 32 bits of mcycle
0xB82	Upper 32 bits of Machine Instructions-Retired (minstreth)	read/write	bit[31:0]: Upper 32 bits of minstret

2.2.1.9. Custom Function Unit Logic Interface (CFU-LI)

Custom Function Unit Logic Interface defines a set of hardware logic signal interfaces which enable you to connect CPUs and CFUs easily. Custom Function Unit (CFU) is a kind of light-weight and customized arithmetic accelerator. With the support of CFU-LI, you can integrate CFUs into your SoC and insert Custom Functions (CF) to deploy CFU hardware, according to actual solution demand.

In the CFU-LI system, the CPU is the controller and the CFU is the responder. The CPU sends the CFU a request and eventually receives the CFU response. For each request, there is exactly one response.

The CFU-LI is stratified into separate feature levels:

- L0: combinational;
- L1: fixed latency;
- L2: variable latency;
- L3: reordering.

You can choose an appropriate interface level and design the responder interface of the CFU. For user-friendliness and in compliance with the official spec, the RISC-V core only supports one kind of interface level, L2. It has downward compatibility to support L0 or L1 as well. You can set some signal constant 0 or 1 to degrade L2 to L1 or L0.

The RISC-V MC core is a -Zicfu compatible core, with a mcfu_selector CSR added and can repurpose three custom function instruction formats. To deploy the resource of CFU, you only need two steps: interface multiplexing and executing CF instructions.

1. The first step is interface multiplexing, which requires writing a specific selector value to mcfu_selector CSR 0xBC0 to select the active CFU and state context.

   The mcfu_selector CSR 0xBC0 has the following fields:

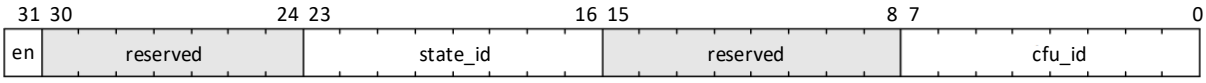

Figure 2.3. mcfu_selector CSR 0xBC0

   - en: enable custom interface multiplexing.
     - When en=0, disable custom interface multiplexing. The cfu_id and state_id fields are ignored. No CFU is selected. The execution of custom-0, custom-1, or custom-2 instructions triggers Exception 2, illegal instruction.
     - When en=1, enable custom interface multiplexing. The cfu_id field selects the current CFU. Custom-0/-1/-2 instructions issue CFU requests to the CFU identified by cfu_id.
   - state_id: select the corresponding state context of the hart's current CFU.
     - This step prepares the CPU for the next step, issuing custom instruction. According to the configuration of selector, the CPU routes the corresponding CFU and its state context to execute subsequent sequences of custom instructions.
   - cfu_id: cfu index.
     - In the scope of a system, cfu index identifies a configured interface implemented by a CFU. When one CFU implements multiple configured interfaces, different CFU_IDs identify which CFU must process the request.

2. The second step is the CPU issuing custom function instructions. The specific function of a CF is defined by customers and identified by custom function identifier (CF_ID). Each CFU packages a set of relevant custom functions. Each CF needs to be implemented by the hardware logic in CFU. You can design the CFU, according to specific scenarios.

   In terms of CF instruction format, we reuse three CF formats/major opcodes: custom-0, custom-1, custom-2. These correspond to three different instructions encoding types: R-type, I-type, and flex-type.

   - Custom-0 R-type encoding
     - Assembly instruction: cfu_reg cf_id, rd, rs1, rs2
     - An R-type CF instruction issues a CFU request for a zero-extended 10-bit CF_ID cf_id with two source register operands identified by rs1 and rs2. The CFU response data is written to destination register rd.

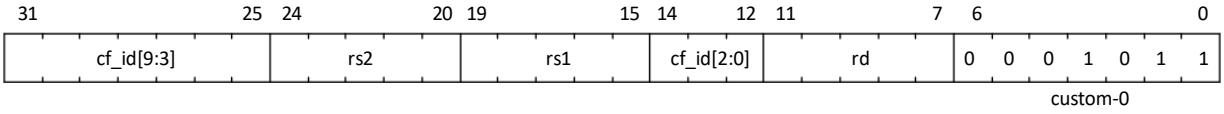

Figure 2.4. CFU R-type Instruction Encoding

   - Custom-1 I-type encoding
     - Assembly instruction: cfu_imm cf_id, rd, rs1, imm
     - An I-type CF instruction issues a CFU request for a zero-extended 4-bit CF_ID cf_id with one source register operand identified by rs1 and a signed-extended 8-bit immediate value imm. The CFU response is written to destination register rd.

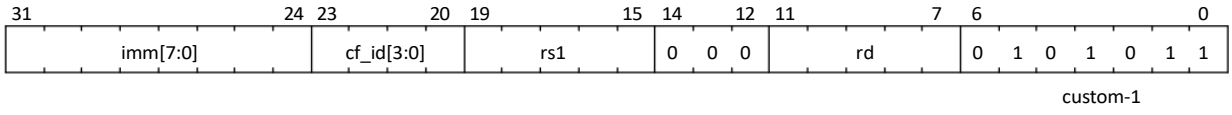

Figure 2.5.CFU I-type Instruction Encoding

- Custom-2 flex-type encoding
  - Assembly instruction: cfu_flex cf_id, rs1, rs2
  - A flex-type CF instruction issues a CFU request for a zero-extended 10-bit CF_ID cf_id with two source register operands identified by rs1 and rs2. There is no destination register and CFU response data is discarded. The instruction is executed purely for its effect upon the selected state context of the selected CFU.

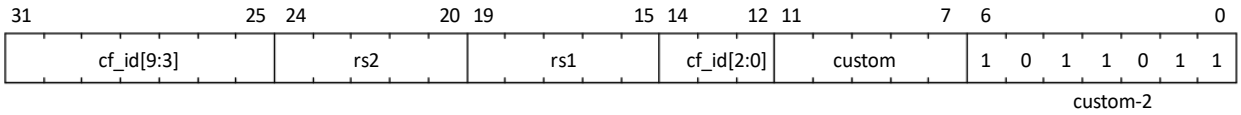

Figure 2.6. CFU Flex-type Instruction Encoding

Alternatively, the cfu_flex25 form of instruction issues an arbitrary 25-bit custom instruction.

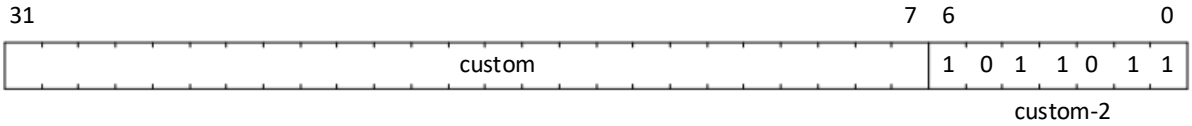

Figure 2.7. CFU Flex-type Instruction Alternate Encoding

A flex-type CF instruction may be used with a CFU-L2 request raw instruction field req_insn to provide an arbitrary 25-bit custom request to a CFU. The absence of an integer destination register field is a feature that provides added, CPU-uninterpreted, custom instruction bits to a CFU.

When the CPU issues a custom instruction, it produces a CFU request which has three sources: the fields of instruction, two source operands from the register file and/or an immediate field of instruction, and the cfu_id and state_id fields of mcfu_selector (Figure 2.3). The CFU request may include the CFU_ID, STATE_ID, raw instruction, CF_ID, and operands. The CFU_ID identifies which CFU must process the request. The CFU includes state context(s) and a data path. The STATE_ID selects the state context to use for this request. The CFU processes the request, possibly updating this state context, and produces a CFU response, which may include the response data. The CPU commits the custom function instruction by writing the response data to the destination register.

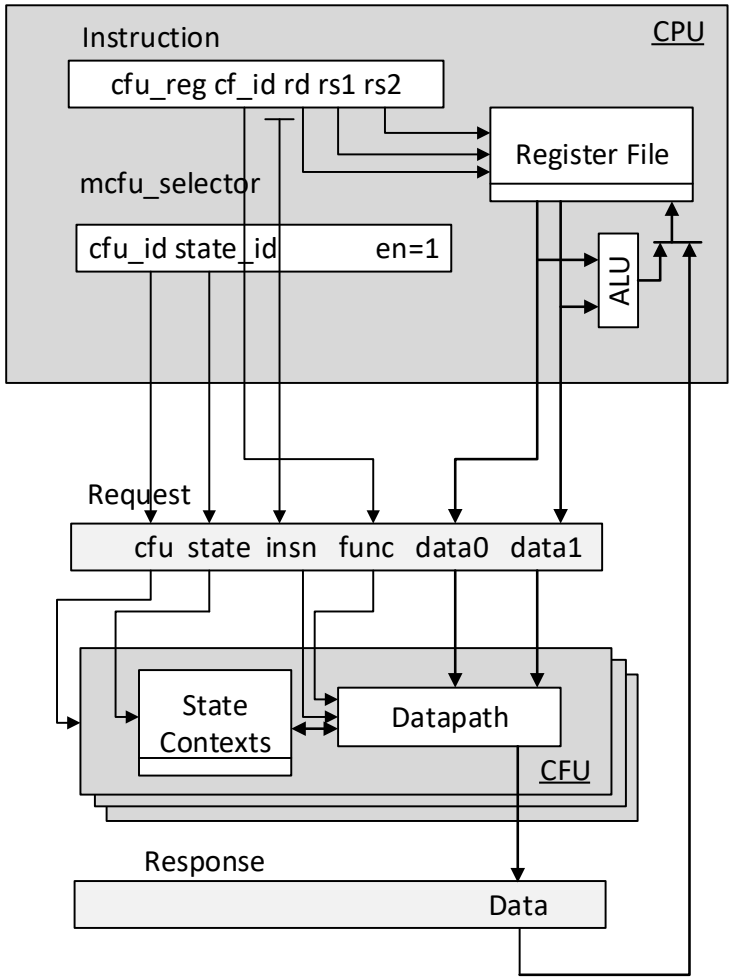

Figure 2.3. Execution of a Custom Function Instruction

Following is a code example illustrating CPU issuing stateful CF instructions f0 and f1 to CFU0, f2 and f3 to CFU1, and f4 to CFU0 again.

```
csrw mcfu_selector,x20 ; select CFU_ID=0 and STATE_ID=HART_ID
cfu_reg 0,x3,x1,x2 ; u0.f0
cfu_reg 1,x6,x5,x4 ; u0.f1
csrw mcfu_selector,x21 ; select CFU_ID=1 and STATE_ID=HART_ID
cfu_reg 2,x9,x7,x8 ; u1.f2
cfu_reg 3,x12,x11,x10 ; u1.f3
csrw mcfu_selector,x20 ; select CFU_ID=0 and STATE_ID=HART_ID again
cfu_reg 4,x15,x13,x14 ; u0.f4
```

1. Write mcfu_selector for CFU_ID=0 and STATE_ID=HART_ID, issue two CF instructions to CFU0.
2. Write mcfu_selector for CFU_ID=1 and STATE_ID=HART_ID, issue two CF instructions to CFU1.
3. Write mcfu_selector for CFU_ID=0 and STATE_ID=HART_ID, issue one CF instruction to CFU0.

2.2.1.10. System Reset Output

The system_resetn_o signal is driven in two ways. When debug is not enabled or if debug reset is not issued, system_resetn_o is the passed value from the input reset signal rst_n_i. It is asynchronous with input clock. When the debugger is enabled and debug reset is issued, the debug reset signal is synchronized to system clock domain and the system_resetn_o is the output of the synchronized signal.

2.2.2. Submodule

The CPU soft IP contains two submodules: PIC and Timer. The PIC and Timer share the same start address in the memory map and a fixed 2 KB address range is allocated, if either PIC or Timer is enabled.

2.2.2.1. PIC

The PIC aggregates up to eight external interrupt inputs (IRQs) into one interrupt output to the processor core. The interrupt status register can be used to read the values of IRQs. Individual IRQs can be configured by programming the corresponding PIC_STATUS, PIC_ENABLE, PIC_SET, and PIC_POL registers. All registers can be accessed through the CPU's internal AHB-L interface, as shown in Figure 2.4.

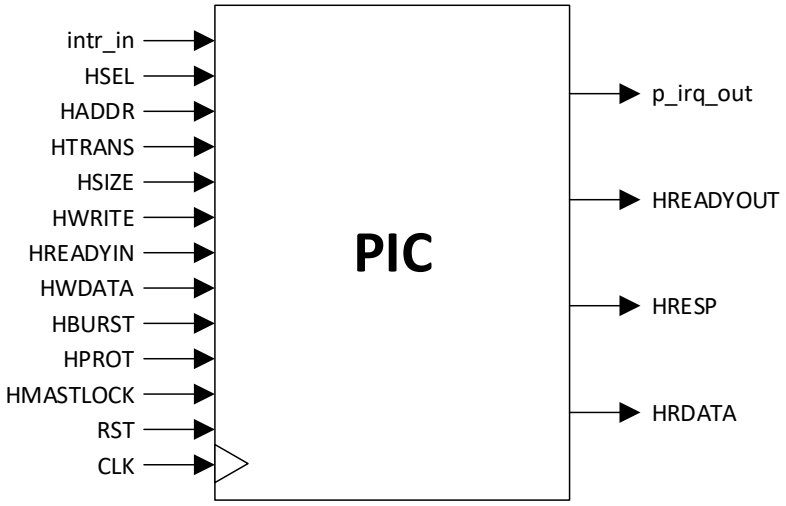

Figure 2.4. PIC Block Diagram

Table 2.2 provides the description of PIC registers.

Table 2.2. PIC Registers

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x000</td><td>PIC_STATUS</td><td>Interrupt Status Register
Access: read-write
Parameterizable width: min=2, max=8
Indicates the pending interrupt at corresponding interrupt request port(irq[i] at top level).

<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[N-1]</td><td>PIC_STATUS [N-1]</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>[1]</td><td>PIC_STATUS [1]</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>[0]</td><td>PIC_STATUS [0]</td><td>RW</td><td>1</td><td>0x0</td></tr>
</table>

PIC_STATUS[i]:
Read
• 0 – no interrupt at irq[i]
• 1 – interrupt pending at irq[i]</td></tr>
</table>

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td></td><td></td><td>Write
• 0 – no effect
• 1 – clear interrupt status for irq[i]</td></tr>
<tr><td>0x004</td><td>PIC_ENABLE</td><td>Interrupt Enable Register
Access: read-write
Parameterizable width: min=2, max=8
Indicates whether the processor responds to the interrupt from corresponding interrupt request port (irq[i]) or not.

<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[N-1]</td><td>PIC_ENABLE[N-1]</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>[1]</td><td>PIC_ENABLE[1]</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>[0]</td><td>PIC_ENABLE[0]</td><td>RW</td><td>1</td><td>0x0</td></tr>
</table>

PIC_ENABLE[i]:
Read
• 0 – irq[i] disabled
• 1 – irq[i] enabled
Write
• 0 – disable irq[i]
• 1 – enable irq[i]</td></tr>
<tr><td>0x008</td><td>PIC_SET</td><td>Interrupt Set Register
Access: write-only
Parameterizable width: min=2, max=8
Sets the interrupt status for corresponding interrupt request port(irq[i]).

<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[N-1]</td><td>PIC_SET [N-1]</td><td>W</td><td>1</td><td>0x0</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>[1]</td><td>PIC_SET [1]</td><td>W</td><td>1</td><td>0x0</td></tr>
<tr><td>[0]</td><td>PIC_SET [0]</td><td>W</td><td>1</td><td>0x0</td></tr>
</table>

PIC_SET[i]:
Read
• Invalid operation gets 0.
Write
• 0 – no effect
• 1 – set interrupt status for irq[i] (set PIC_STATUS[i])</td></tr>
</table>

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x00C</td><td>PIC_POL</td><td>Interrupt Polarity Register
Access: read-write
Parameterizable width: min=2, max=8
Indicates the polarity of corresponding interrupt request (irq[i]) port.

<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[N]</td><td>PIC_POL [N]</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>[1]</td><td>PIC_POL l[1]</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>[0]</td><td>PIC_POL [0]</td><td>RW</td><td>1</td><td>0x0</td></tr>
</table>

PIC_POL[i]:
Read
• 0 – irq[i] is active high
• 1 – irq[i] is active low
Write
• 0 – Set irq[i] active high
• 1 – Set irq[i] active low</td></tr>
</table>

Note: The register definition of PIC follows Lattice Interrupt Interface (LINTR) Standard, refer to Lattice Memory Mapped Interface and Lattice Interrupt Interface User Guide (FPGA-UG-02039) for more information.

2.2.2.2. Timer

The Timer module provides a 64-bit real-time counter register, mtime, and time compare register, mtimecmp. An output interrupt signal notifies the RISC-V processor core when the value of mtime is greater than or equal to the value of mtimecmp. All registers can be accessed through the CPU's internal AHB-L interface, as shown in Figure 2.5.

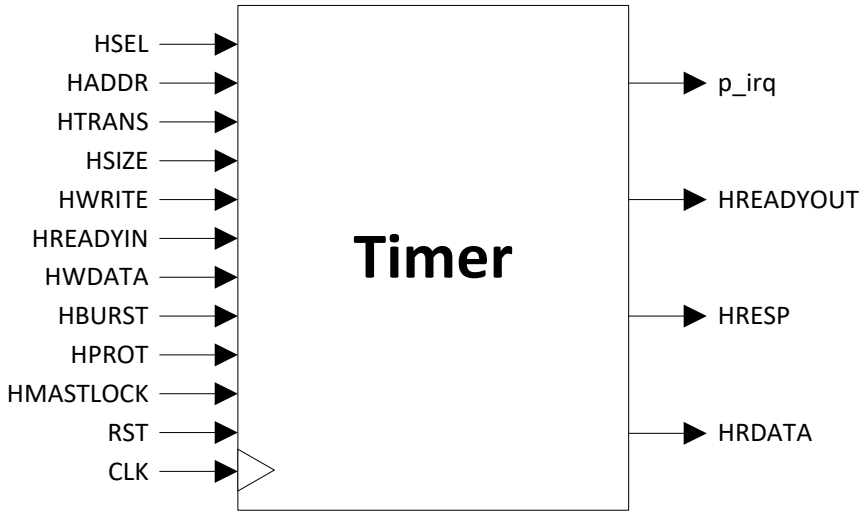

Figure 2.5. Timer Block Diagram

Table 2.3 provides the description of Timer registers.

Table 2.3. Timer Registers

Offset	Name	Description
0x400	TIMER_CNT_L	Lower 32 bits of Timer counter register. Field | Name | Access | Width | Reset [63:0] | mtime | RW | 64 | 0x0 mtime A 64-bit real-time counter register. You must set the register to a non-zero value to start the counting process.
0x404	TIMER_CNT_H	Higher 32 bits of Timer counter register.
0x410	TIMER_CMP_L	Lower 32 bits for Timer time compare register. Field | Name | Access | Width | Reset [63:0] | mtimecmp | RW | 64 | 0x0 mtimecmp This register is used to generate or clear the timer interrupt, mtip. When the value of mtime register is greater than or equal to the value of mtimecmp register, cpu_mtip_o is asserted. The interrupt remains posted until mtimecmp becomes greater than mtime, typically as a result of writing mtimecmp.
0x400	TIMER_CNT_L	Higher 32 bits for Timer time compare register.

2.3. Signal Description

Table 2.4 to Table 2.7 list the ports of the CPU soft IP in different categories.

2.3.1. Clock and Reset

Table 2.4. Clock and Reset Ports

Name	Direction	Width	Description
clk_i	In	1	RISC-V soft IP clock
rst_n_i	In	1	Global reset, active low
system_resetn_o	Out	1	Combined Global reset and Debug Reset from JTAG, active low

2.3.2. Instruction and Data Interface

Table 2.5. Instruction Ports

Name	Direction	Width	Description
AHBL_M0_INSTR - HADDR	Out	32	—
AHBL_M0_INSTR - HWRITE	Out	1	Fixed to 1'b0
AHBL_M0_INSTR - HSIZE	Out	3	Fixed to 3'b010
AHBL_M0_INSTR - HPROT	Out	4	Fixed to 4'b1110 when caches are not enabled.
AHBL_M0_INSTR - HTRANS	Out	2	—
AHBL_M0_INSTR - HBURST	Out	3	—
AHBL_M0_INSTR - HMASTLOCK	Out	1	Fixed to 1'b0
AHBL_M0_INSTR - HWDATA	Out	32	—

Name	Direction	Width	Description
AHBL_M0_INSTR - HRDATA	In	32	—
AHBL_M0_INSTR - HREADY	In	1	—
AHBL_M0_INSTR - HRESP	In	1	—

Table 2.6. Data Ports

Name	Direction	Width	Description
AHBL_M1_DATA - HADDR	Out	32	—
AHBL_M1_DATA - HWRITE	Out	1	—
AHBL_M1_DATA - HSIZE	Out	3	—
AHBL_M1_DATA - HPROT	Out	4	Fixed to 4'b1111 when caches are not enabled.
AHBL_M1_DATA - HTRANS	Out	2	—
AHBL_M1_DATA - HBURST	Out	3	—
AHBL_M1_DATA - HMASTLOCK	Out	1	—
AHBL_M1_DATA - HSEL	Out	1	—
AHBL_M1_DATA - HWDATA	Out	32	—
AHBL_M1_DATA - HRDATA	In	32	—
AHBL_M1_DATA - HREADY	In	1	—

Note: Refer to AMBA 3 AHB-Lite Protocol V1.0 for more information.

2.3.3. Interrupt Interface

Table 2.7. Interrupt Ports

Name	Type	Width	Description
IRQ_Sx	In	1~8	Peripheral interrupts.
TIMER_IRQ_M0	Out	1	Timer interrupt output, exists only when TIMER_ENABLE is checked.
TIMER_IRQ_S0	In	1	Timer interrupt input, exists only when TIMER_ENABLE is unchecked.

2.3.4. CFU-LI Interface

CFU-LI is used to connect CFU accelerator. RISC-V core supports two CFU-LIs. You can enable CFU-LI and configure the number of CFU-LI in the Module/IP Block Wizard GUI.

Table 2.8. CFU-LI Ports (Optional)

Port	Direction	Width	Group	Description
req_valid	out	1	Request	Request valid
req_ready	in	1		Request ready
req_cfu	out	4		Request CFU_ID
req_state	out	3		Request STATE_ID
req_func	out	3		Request CF_ID
req_insn	out	32		Request raw instruction
req_data0	out	32		Request operand data 0
req_data1	out	32		Request operand data 1
resp_valid	in	1	Response	Response valid
resp_ready	out	1		Response ready
resp_status	in	3		Response status
resp_data	in	32		Response data

2.4. Attribute Summary

The configurable attributes of the RISC-V MC CPU IP are shown in Table 2.9. and are described in Table 2.10.

The attributes can be configured through the Lattice Propel Builder software.

Table 2.9. Configurable Attributes

Attribute	Selectable Values	Default	Dependency on Other Attributes
General			
CACHE_ENABLE	Checked, Unchecked	Checked	—
ICACHE_RANGE_LOW	0~0xFFFFFFFF	0	Enabled when CACHE_ENABLE
ICACHE_RANGE_HIGH	0~0xFFFFFFFF	0xF0000000	Enabled when CACHE_ENABLE
DCACHE_RANGE_LOW	0~0xFFFFFFFF	0	Enabled when CACHE_ENABLE
DCACHE_RANGE_HIGH	0~0xFFFFFFFF	0xF0000000	Enabled when CACHE_ENABLE
DEBUG_ENABLE	Checked, Unchecked	Checked	—
SOFT_JTAG	Checked, Unchecked	Checked when DEBUG_ENABLE and DEVICE == “LAV-AT”	—
TIMER_ENABLE	Checked, Unchecked	Checked	—
PIC_ENABLE	Checked, Unchecked	Checked	—
PICTIMER_START_ADDR	0~0xFFFFFB00	0xFFFF0000	Enabled when PIC_ENABLE, or TIMER_ENABLE
IRQ_NUM	2~8	8	Enabled when PIC_ENABLE
C_EXT	Checked, Unchecked	Checked	—
M_EXT	Checked, Unchecked	Unchecked	Enabled when C_EXT For Certus-NX, CertusPro-NX, CrossLink-NX, and MachXO5-NX devices only
JTAG_CHANNEL	14~16	14	Enabled when DEBUG_ENABLE
CFU_EN	Checked, Unchecked	Unchecked	—
CFU_N_CFUS	1,2	1	Enabled when CFU_EN

Table 2.10. Attributes Description

Attribute	Description
ICACHE_ENABLE	1: enable instruction cache 0: disable instruction cache
DCACHE_ENABLE	1: enable data cache 0: disable data cache
ICACHE_RANGE_LOW	Lower limit of cacheable address range for instruction cache, this address itself is included.
ICACHE_RANGE_HIGH	Higher limit of cacheable address range for instruction cache, this address itself is included.
DCACHE_RANGE_LOW	Lower limit of cacheable address range for data cache, this address itself is included.
DCACHE_RANGE_HIGH	Higher limit of cacheable address range for data cache, this address itself is included.
DEBUG_ENABLE	1: debug function enable 0: debug function disable
SOFT_JTAG	1: debug with hard JTAG 0: debug with soft JTAG
TIMER_ENABLE	1: timer enable 0: timer disable
PIC_ENABLE	1: pic enable 0: pic disable
PICTIMER_START_ADDR	Start address of PIC and Timer
IRQ_NUM	Number of Peripheral Interrupts

Attribute	Description
C_EXT	1: support for compressed instruction 0: no support for compressed instruction
M_EXT	1: support for M standard extension 0: no support for M standard extension
JTAG_CHANNEL	JTAG channel select
CFU_EN	Enable CFU. When CFU is enabled, cache and debug cannot be edited.
CFU_N_CFUS	CFU interface number

Notes:

- ICACHE_ENABLE and DCACHE_ENABLE should only be enabled when using CrossLink-NX devices with sufficient resources. For MachXO2, MachXO3L, MachXO3LF, and MachXO3D devices, cache features are not supported as their resources are limited.
- ICACHE_RANGE_LOW should not be larger than ICACHE_RANGE_HIGH, and address range between ICACHE_RANGE_LOW and ICACHE_RANGE_HIGH should not be overlapped with peripheral device address ranges. The memory which stores instructions should always be fixed in this range.
- Similarly, DCACHE_RANGE_LOW should not be larger than DCACHE_RANGE_HIGH, and address range between DCACHE_RANGE_LOW and DCACHE_RANGE_HIGH should not be overlapped with peripheral devices address ranges.

3. RISC-V MC CPU IP Generation

This section provides information on how to generate the CPU IP Core module using Lattice Propel Builder.

To generate the CPU IP Core:

1. In Lattice Propel Builder, create a new design. Select the CPU package. Enter the component name, as shown in Figure 3.1. Click **Next**.

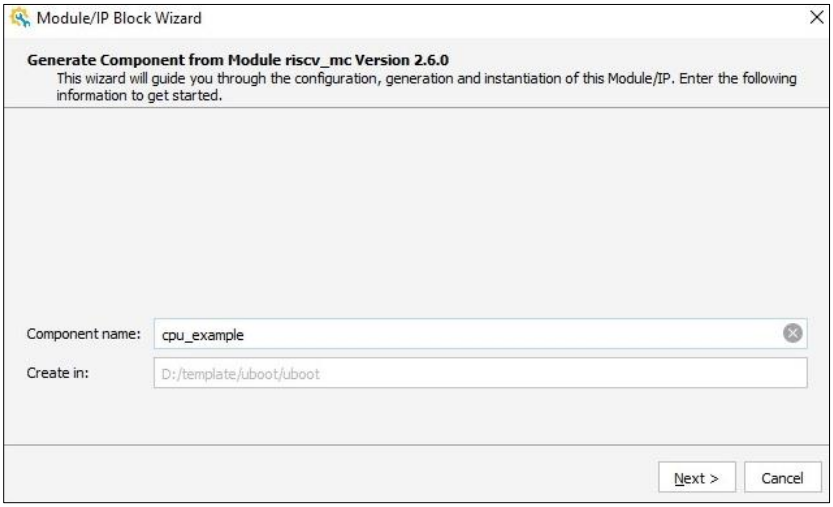

Figure 3.1. Entering Component Name

2. Configure the parameters, as shown in Figure 3.2. Click **Generate**.

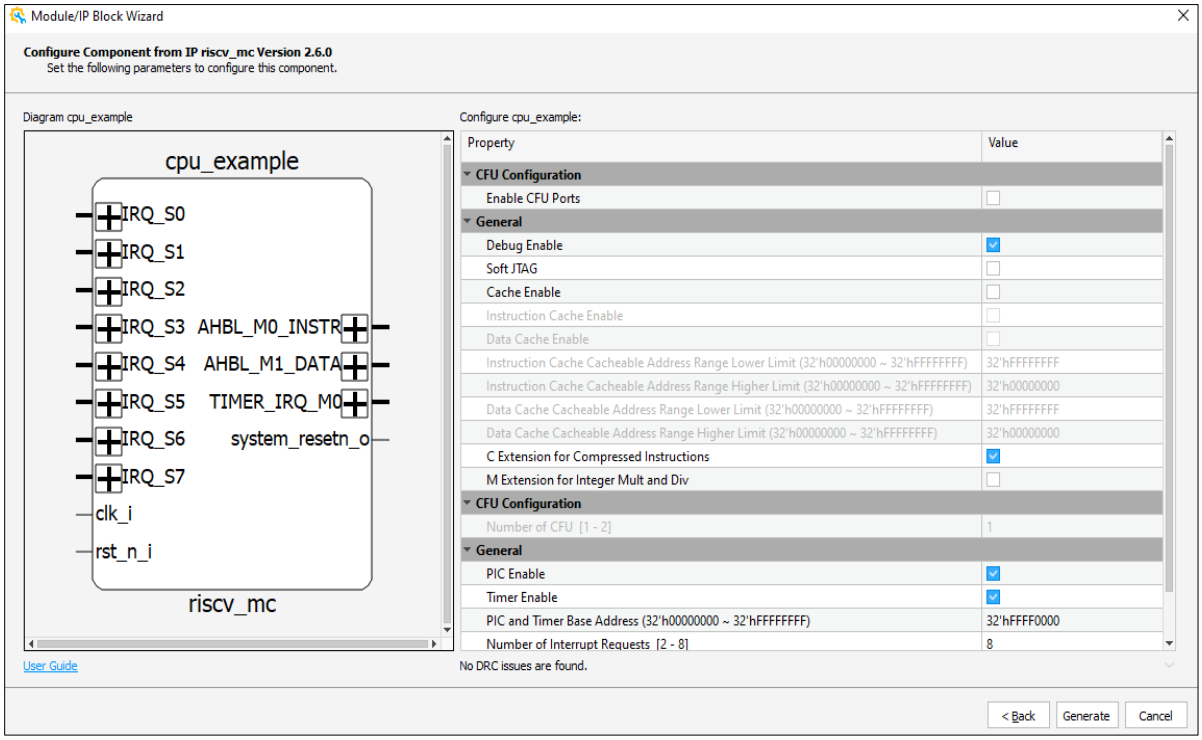

Figure 3.2. Configuring Parameters

3. Verify the information. Click **Finish**.

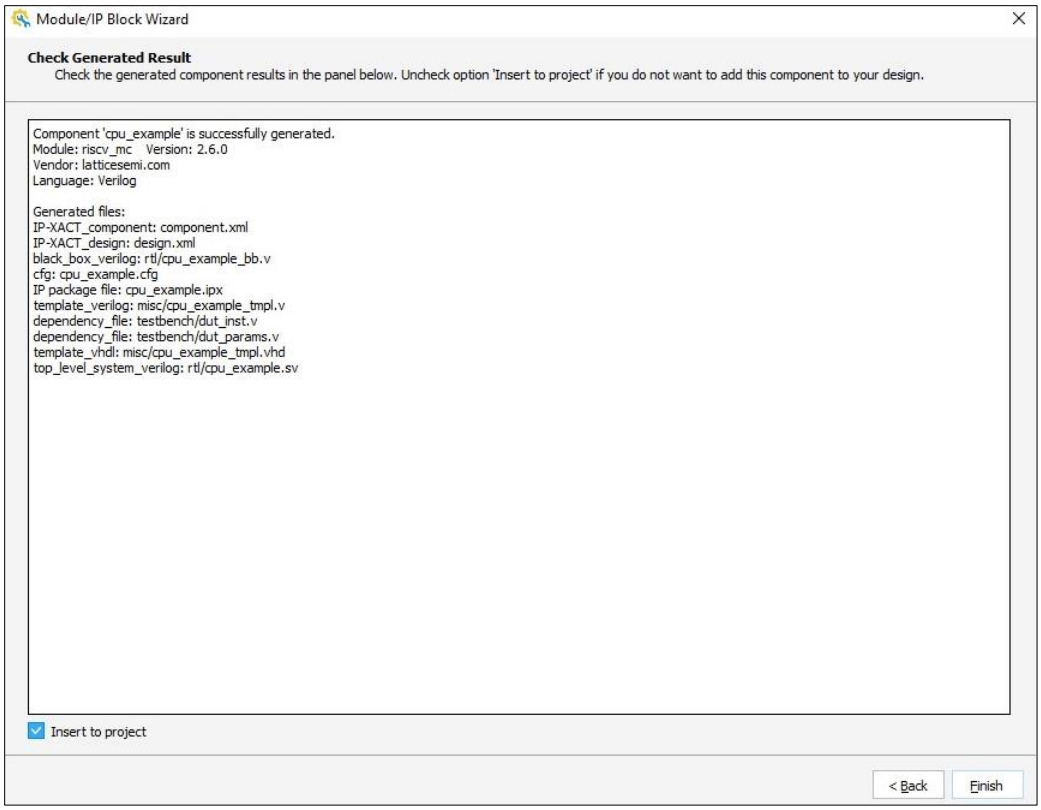

Figure 3.3. Verifying Results

4. Confirm or modify the module instance name. Click **OK**.

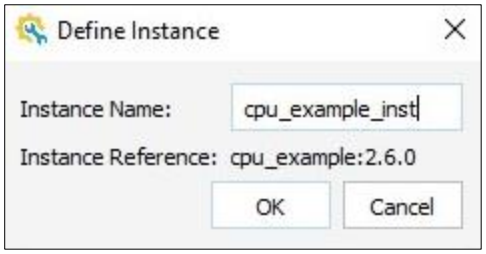

Figure 3.4. Specifying Instance Name

The CPU IP instance is successfully generated, as shown in Figure 3.5.

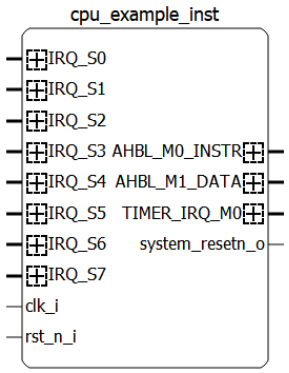

Figure 3.5. Generated Instance

Appendix A. Resource Utilization

Table A.1. Resource Utilization in MachXO3D Device (with Cache Disabled)

Configuration	LUTs	Registers	EBRs
Processor core only	1450	806	4
Processor core + PIC	1559	845	4
Processor core + Timer	1844	955	4
Processor core + Debug	1726	1108	4
Processor core + C_EXT	1738	860	4
Processor core + PIC + Timer	1927	989	4
Processor core + PIC + Timer + Debug	2162	1287	4
Processor core + PIC + Timer + Debug + C_EXT	2385	1345	4

Note: Resource utilization characteristics are generated using Lattice Diamond software.

Table A.2. Resource Utilization in CrossLink-NX Device (with Cache Disabled)

Configuration	LUTs	Registers	EBRs	DSP
Processor core only	1620	821	2	0
Processor core + PIC	1789	859	2	0
Processor core + Timer	2103	959	2	0
Processor core + Debug	1979	1194	2	0
Processor core + C_EXT	1911	830	2	0
Processor core + C_EXT + M_EXT	2417	1172	2	6
Processor core + PIC + Timer	2226	994	2	0
Processor core + PIC + Timer + Debug	2494	1375	2	0
Processor core + PIC + Timer + Debug + C_EXT	2790	1450	2	0

Note: Resource utilization characteristics are generated using Lattice Radiant software.

Table A.3. Resource Utilization in CrossLink-NX Device (with Cache Enabled)

Configuration	LUTs	Registers	EBRs	DSP
Processor core + only	3372	1423	16	0
Processor core + PIC	3568	1479	16	0
Processor core + Timer	3753	1574	16	0
Processor core + Debug	3687	1810	16	0
Processor core + C_EXT	3711	1551	16	0
Processor core + C_EXT + M_EXT	4133	1879	16	6
Processor core + PIC + Timer	3894	1606	16	0
Processor core + PIC + Timer + Debug	4309	1980	16	0
Processor core + PIC + Timer + Debug + C_EXT	4568	2032	16	0

Note: Resource utilization characteristics are generated using Lattice Radiant software.

Table A.4. Resource Utilization in Lattice Avant Device (with Cache Disabled)

Configuration	LUTs	Registers	EBRs	DSP
Processor core + only	2015	867	2	0
Processor core + PIC	2097	886	2	0
Processor core + Timer	2456	979	2	0
Processor core + Debug	2376	1268	2	0
Processor core + C_EXT	2202	860	2	0
Processor core + C_EXT + M_EXT	2731	1284	2	6
Processor core + PIC + Timer	2535	1017	2	0
Processor core + PIC + Timer + Debug	2869	1444	2	0
Processor core + PIC + Timer + Debug + C_EXT	3128	1528	2	0

Note: Resource utilization characteristics are generated using Lattice Radiant software.

Table A.5. Resource Utilization in Lattice Avant Device (with Cache Enabled)

Configuration	LUTs	Registers	EBRs	DSP
Processor core + only	3340	1364	18	0
Processor core + PIC	3437	1406	18	0
Processor core + Timer	3857	1548	18	0
Processor core + Debug	3708	1806	18	0
Processor core + C_EXT	3671	1546	18	0
Processor core + C_EXT + M_EXT	4431	1861	18	6
Processor core + PIC + Timer	3809	1655	18	0
Processor core + PIC + Timer + Debug	4381	1995	18	0
Processor core + PIC + Timer + Debug + C_EXT	4559	2168	18	0

Note: Resource utilization characteristics are generated using Lattice Radiant software.

Table A.6. Resource Utilization in CertusPro-NX Device (with Cache Disabled)

Configuration	LUTs	Registers	EBRs	DSP
Processor core + only	1978	834	2	0
Processor core + PIC	2105	886	2	0
Processor core + Timer	2446	990	2	0
Processor core + Debug	2304	1218	2	0
Processor core + C_EXT	2288	896	2	0
Processor core + C_EXT + M_EXT	2790	1185	2	6
Processor core + PIC + Timer	2578	1032	2	0
Processor core + PIC + Timer + Debug	2871	1390	2	0
Processor core + PIC + Timer + Debug + C_EXT	3220	1461	2	0

Note: Resource utilization characteristics are generated using Lattice Radiant software.

Table A.7. Resource Utilization in CertusPro-NX Device (with Cache Enabled)

Configuration	LUTs	Registers	EBRs	DSP
Processor core + only	3031	1461	20	0
Processor core + PIC	3391	1505	20	0
Processor core + Timer	3812	1607	20	0
Processor core + Debug	3667	1854	20	0
Processor core + C_EXT	3691	1613	19	0
Processor core + C_EXT + M_EXT	4334	2007	19	6
Processor core + PIC + Timer	3926	1671	20	0
Processor core + PIC + Timer + Debug	4139	2012	20	0
Processor core + PIC + Timer + Debug + C_EXT	4450	2169	19	0

Note: Resource utilization characteristics are generated using Lattice Radiant software.

Appendix B. Debug with Soft JTAG

To debug with Soft JTAG:

1. In Lattice Propel Builder, enable **Soft JTAG** in the Module/IP Block Wizard GUI (Figure 3.2) when generating the IP.
2. After the IP is generated, right-click on the **JTAG** port and select **Export** (Figure B.1).

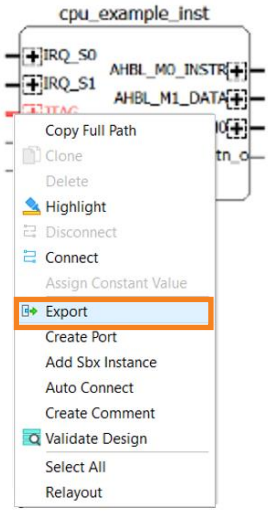

Figure B.1. Exporting Pins

3. Assign the normal I/O as JTAG I/O using the Device Constraint Editor in Lattice Radiant software.
   a. Synthesize the design SoC in Lattice Radiant software by clicking **Synthesis Design** from the process toolbar.
   b. Open **Device Constraint Editor** from the **Tools** tab in Lattice Radiant software and assign the pins. For different devices, refer to the user guide of each board. The following assignment is for LFCPNX-100-9LF672C (Figure B.2).

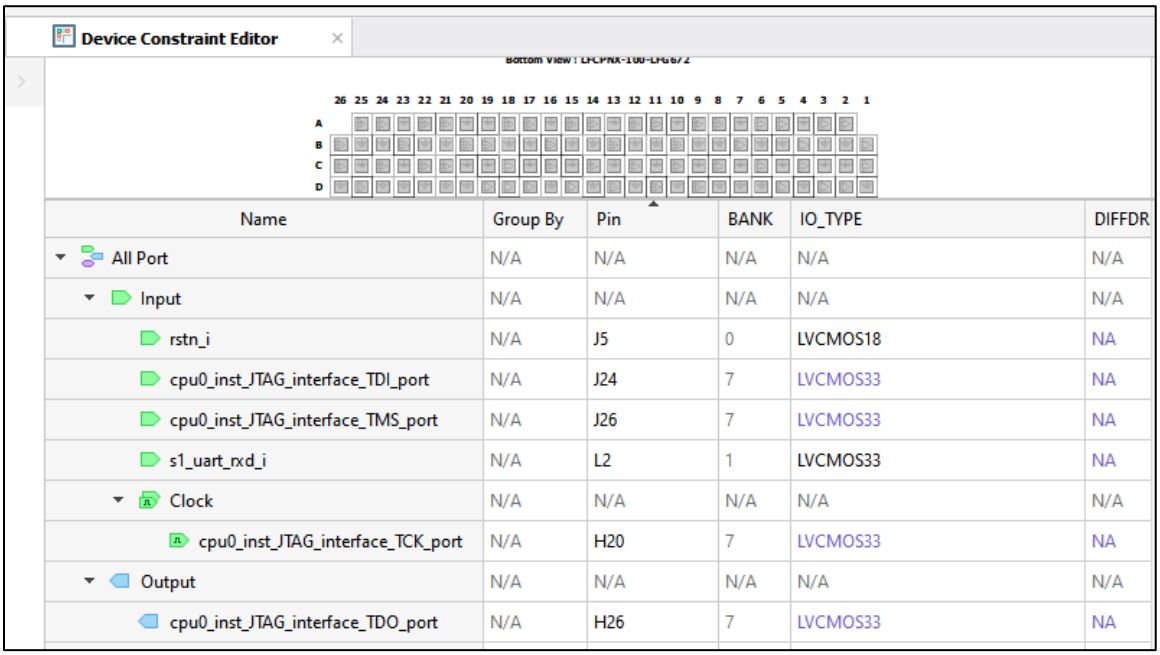

Figure B.2. Assigning Pins

c. Double-click on the targeted strategy in the **File List** view to open the **Strategies** dialog box.

d. In the **Strategies** dialog box, set the environment variable for **Place & Route Design**. Enter "-exp WARNING_ON_PCLKPLC1=1" in the Value of **Command Line Options** if TCK connects to normal I/O (Figure B.3).

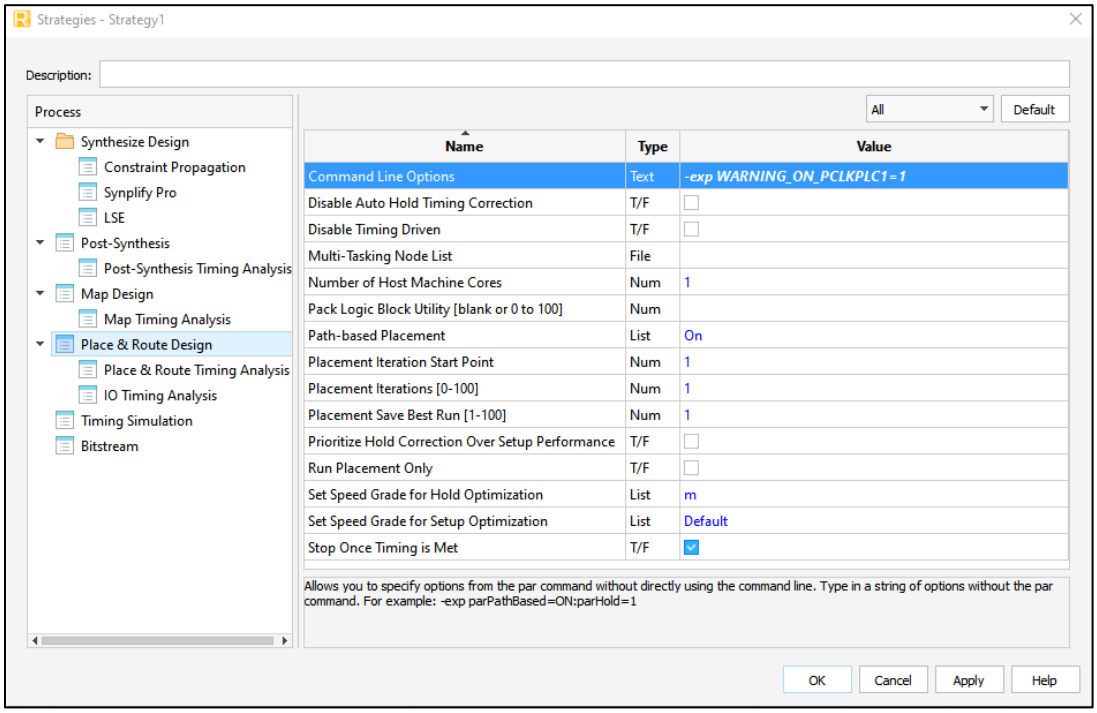

Figure B.3. Setting Environment Variables

e. Generate the bitstream and load it to the board.

f. Connect the pins on cable to the board according to your assignments. Connect VCC and GND. Scan the cable in Lattice Propel SDK software and ignore the scanning of the device.
Note: C projects generated for Lattice Avant family devices cannot use Soft JTAG to debug on MachXO5-NX, Certus-NX, CertusPro-NX, and CrossLink-NX boards and vice versa.

References

- Lattice Propel 2024.1 Builder User Guide (FPGA-UG-02212)
- AMBA 3 AHB-Lite Protocol V1.0
- RISC-V Privileged Specification (20211203)
- RISC-V Instruction Set Manual (20190608)
- Lattice Memory Mapped Interface and Lattice Interrupt Interface User Guide (FPGA-UG-02039)

For more information, refer to:

- Lattice Propel web page
- Lattice Avant-E Family Devices web page
- MachXO5-NX Family Devices web page
- Certus-NX Family Devices web page
- CertusPro-NX Family Devices web page
- CrossLink-NX Family Devices web page
- MachXO3D Family Devices web page
- MachXO3 Family Devices web page
- MachXO2 Family Devices web page
- ECP5 & ECP5-5G Family Devices web page
- Lattice Insights for Lattice Semiconductor Training Series and Learning Plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.0, May 2024

Section	Change Summary
All	Production release.

www.latticesemi.com