# RISC-V RX CPU IP Core

# User Guide

FPGA-IPUG-02241-1.0

October 2023

## 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 REDUDANCIES, 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

# Acronyms in This Document

A list of acronyms used in this document.

| Acronyms | Definition |
|---|---|
| ABI | Application Binary Interface |
| AEE | Application Execution Environment |
| AXI | Advanced eXtensible Interface |
| CF | Custom Function |
| CFU | Custom Function Unit |
| CFU-LI | Custom Function Unit Logic Interface |
| CLINT | Core Local Interrupter |
| CPU | Central Processing Unit |
| CSR | Control and Status Register |
| DDR | Double Data Rate |
| DMIPS | Dhrystone MIPS (Million Instructions per Second) |
| DTCM | Data TCM |
| IP | Intellectual Property |
| EIP | External Interrupt Pending |
| FPGA | Field Programmable Gate Array |
| GDB | Gnu Debugger |
| GPIO | General Purpose Input/Output |
| GUI | Graphical User Interface |
| HDL | Hardware Description Language |
| IE | Interrupt Enable |
| IRQ | Interrupt Request |
| ISA | Instruction Set Architecture |
| JTAG | Joint Test Action Group |
| ITCM | Instruction TCM |
| LUT | Lookup-Table |
| NMI | Non-Maskable Interrupt |
| OpenOCD | Open On-Chip Debugger |
| OS | Operating System |
| PC | Personal Computer |
| PLIC | Platform-Level Interrupt Controller |
| PMP | Physical Memory Protection |
| RISC-V | Reduced Instruction Set Computer-V (five) |
| RV32IMC | RISC-V Integer, M & Compressed Instruction Sets |
| RX | Real Time OS (RISC-V for RTOS applications) |
| SDRAM | Synchronous Dynamic Random-Access Memory |
| SEE | Supervisor Execution Environment |
| SoC | System-on-Chip |
| TCM | Tightly Coupled Memory |
| UART | Universal Asynchronous Receiver Transmitter |
| WARL | Write Any Values, Reads Legal Values |
| WDT | Watchdog Timer Device |

# 1. Introduction

The Lattice Semiconductor RISC-V RX soft IP contains a 32-bit RISC-V processor core and several submodules – Platform Level Interrupt Controller (PLIC), Core Local Interrupter (CLINT), and Watchdog. The CPU core supports the RV32IMC instruction set and debug feature which is JTAG – IEEE 1149.1 compliant. The modules outside are accessed by the processor core using AXI or Local Bus Interface.

The design is implemented in Verilog HDL. It can be configured and generated using the Lattice Propel™ Builder software. It is targeted for Lattice Avant™, MachXO5™-NX, CrossLink™-NX, Certus™-NX, and CertusPro™-NX FPGA devices. The design is implemented using Lattice Radiant™ software Place and Route tool integrated with the Synplify Pro® synthesis tool.

## 1.1. Quick Facts

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

**Table 1.1 RISC-V RX Soft IP Quick Facts**

| | | |
|---|---|---|
| **IP Requirements** | Supported FPGA Families | Lattice Avant, MachXO5-NX, CrossLink-NX, Certus-NX, CertusPro-NX |
| **Resource Utilization** | Targeted Devices | LAV-AT, LFMXO5, LIFCL, LFD2NX, LFCPNX |
| | Supported User Interfaces | AXI Interface, Local Bus Interface |
| | Resources | See Table A.1 and Table A.2. |
| **Design Tool Support** | Lattice Implementation | Lattice Propel Builder 2023.2, Lattice Radiant 2023.2 |
| | Simulation | For a list of supported simulators, see the Lattice Radiant and Lattice Diamond™ software user guide. |

## 1.2. Features

- The RISC-V RX soft IP has the following features:
- RV32IMC instruction set
- Five stage pipeline
- All three privilege modes supported: Machine mode, Supervisor mode, and User mode
- Instruction Cache and Data Cache
- Support for the AXI4 bus standard for data port
- Debug through Gnu Debugger (GDB) and Open On-Chip Debugger (OpenOCD)
- PLIC module
- CLINT module
- Watchdog module
- Benchmark and Frequency:
  - Balanced mode: 1.01 DMIPS/MHz performance; 130 MHz (sp9)/110 MHz (sp7) on CertusPro-NX device

  **Note**: $f_{max}$ is based on:
  - Standalone processor core
  - Radiant 2023.1 production build, with 9_High-Performance_1.0V (sp9) and 7_High[1]Performance_1.0V (sp7)

## 1.3. Conventions

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

# 2. Functional Descriptions

## 2.1. Overview

The RISC-V RX 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 submodules. Among submodules, PLIC and CLINT/Watchdog are required, while Local UART is optional. The AXI Instruction Port and both TCM ports are also optional.

The 32-bit processor can use the AXI Instruction Port or the local instruction port to fetch instructions from an external AXI device or a TCM, respectively. The processor can use the AXI Data Port or the local data port to access data. Among these AXI and local bus ports, the AXI Instruction Port and both TCM local bus ports, as shown in Figure 2.1, are optional in the RX configuration dialog. But either the AXI Instruction Port or both of the TCM ports must be enabled to make the RX core perform normally.

The CPU core, bridges, MUX, PLIC, and UART run in the fast system clock domain. The CLINT and the Watchdog run in both the fast system clock domain and the slow real time clock domain. The Debug module runs in both the system clock domain and the JTAG clock domain.

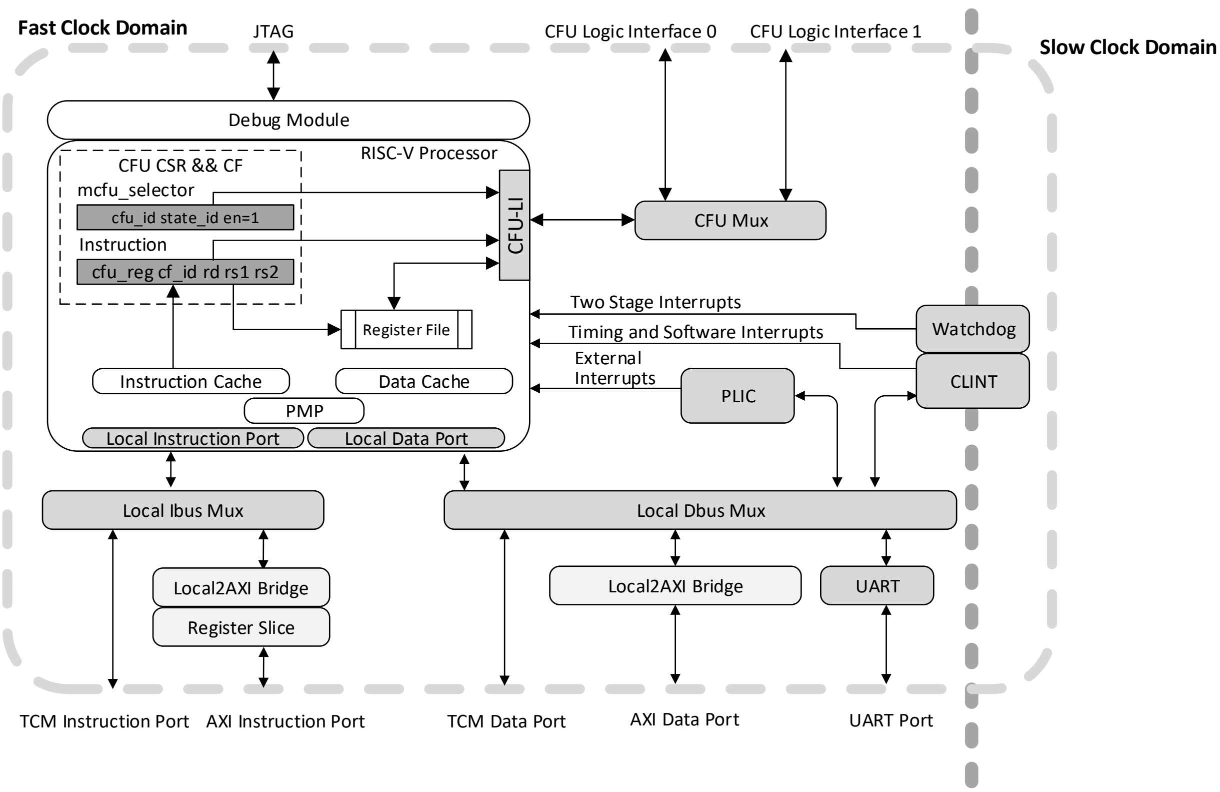


**Figure 2.1. RISC-V RX Soft IP Diagram (with All Features Enabled)**

## 2.2. Modules Description

### 2.2.1. RISC-V Processor Core

Figure 2.2 shows the processor core block diagram.

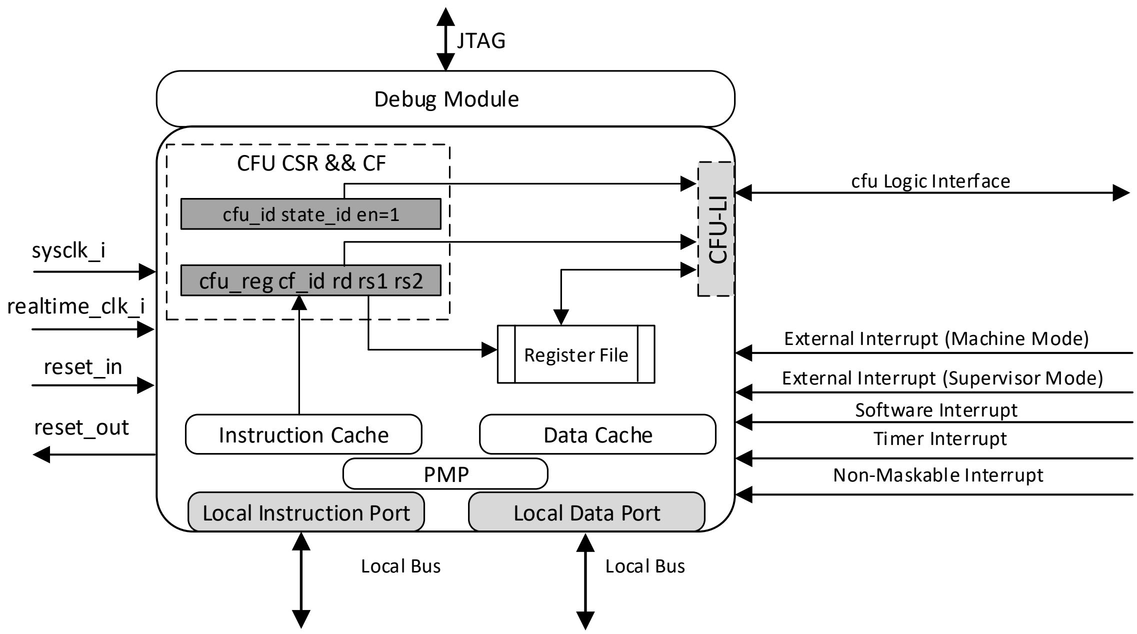


**Figure 2.2. RISC-V RX Processor Core Block Diagram**

#### 2.2.1.1. F Extension Support

Version 2.3 of the RX core supports the F Extension. For more details, refer to the related chapter of RISC-V Privileged Specification (Version 20211203).

#### 2.2.1.2. Interrupt

There are four types of interrupts, External Interrupt from PLIC in Machine mode or Supervisor mode, Software Interrupt, Timer Interrupt from CLINT, and Non-Maskable Interrupt from outside.

- External Interrupt
  - In this version, the RX processor core expands the number of external interrupts to 32 in total, and 30 of them are available to you.
- NMI
  - A basic non-maskable interrupt (NMI) is supported in this version of RX. There is a new Control and Status Register (CSR) named mnvec for you to set specific trap entry for NMI routine. Its CSR address is 0x7C0.
  - There is a new input port nmiInterrupt for the incoming interrupt. When there is an asserted input, the PC jumps to the address stored in mnvec. For other types of interrupts, it jumps to mtvec. Below is an example asm code:

```
#define CSR_MNVEC              0x7C0
…
la t0, trap_entry_nmi
csrw CSR_MNVEC, t0
```

  - The current NMI implementation is non-recoverable. It is expected for the processor to jump into the correct interrupt service, but there is no guarantee for how it gets returned. General interrupt has the mepc CSR register to store the PC address before jumping to the interrupt, so that when the processor returns from it,

mepc is used to restore the previous PC address. NMI does not have such a register. When the processor comes back from NMI, it may go out of control.
Note: there is a task group aiming to define the recoverable NMI, which is still on track, with no stable draft specification yet.

By default, interrupts are handled in Machine mode. Considering Supervisor mode is supported, it is possible to delegate certain interrupts to Supervisor mode.

#### 2.2.1.3. 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.

#### 2.2.1.4. Low Power Mode

The processor core enters low power mode when it executes the WFI instruction. The PC halts during the low power mode. The processor wakes up if there is an external/timer interrupt.

#### 2.2.1.5. Reset Vector

The reset vector is set to 0x0000_0000 and it is fixed.

#### 2.2.1.6. Branch Prediction

The RX core uses dynamic target prediction for branches.

#### 2.2.1.7. 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 in Lattice Avant, MachXO5-NX, CrossLink-NX, CertusPro-NX, and Certus-NX devices. For more information, refer to Appendix B.

**Table 2.1. Soft JTAG Ports**

| Name | Direction | Width | Description |
|---|---|---|---|
| TDI | In | 1 | Test data input pin. |
| TCK | In | 1 | Test data output pin. |
| TMS | In | 1 | Test clock pin. |
| TDO | Out | 1 | Test mode select pin for controlling the TAP state machine. |

#### 2.2.1.8. Cache

Both Instruction Cache and Data Cache have the following configurations:

- cache size: 4096 bytes
- 32 bytes per cache line
- 2-way set associative

The cache strategy for data cache is write through. The cache eviction policy of both caches is round robin. To flush the caches, refer to annotations of cache.h in the driver codes. There are three APIs to be used to flush either the instruction or data cache at the run time.

#### 2.2.1.9. Privilege Mode

The processor supports User, Supervisor, and Machine mode. Along with corresponding CSR registers, Figure 2.3 shows two typical software stacks:

- A simple system that supports only a single application running on an application execution environment (AEE). The application is coded to run with a particular application binary interface (ABI). ABI includes the supported user-level Instruction Set Architecture (ISA) plus a set of ABI calls to interact with the AEE. The ABI hides details of the AEE from the application to allow greater flexibility in implementing the AEE.

- Meanwhile, a conventional operating system (OS) can provide AEE and ABI. The OS interfaces with a supervisor execution environment (SEE) via a supervisor binary interface (SBI). An SBI comprises the user-level and supervisor-level ISA together with a set of SBI function calls.

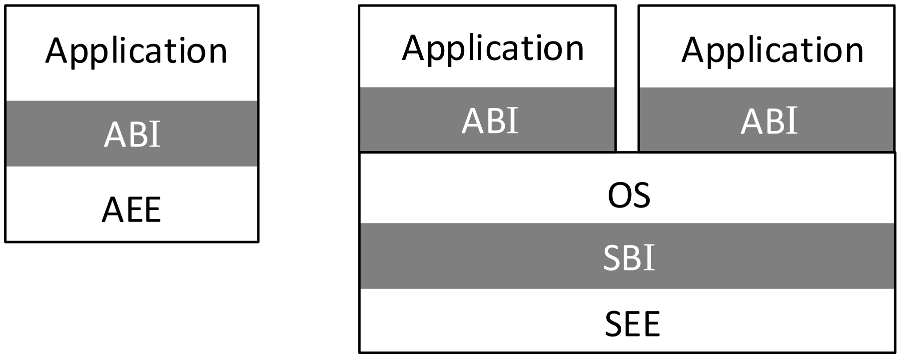


**Figure 2.3. Various Forms of Privileged Execution**

#### 2.2.1.10. Control and Status Registers

The processor core supports three privilege modes. All supported Control and Status Registers (CSRs) are listed in Table 2.2.

**Table 2.2. RISC-V Processor Core Control and Status Registers**

| Number | Privilege | Name | Description |
|---|---|---|---|
| **Supervisor Trap Setup** | | | |
| 0x100 | SRW | sstatus | Supervisor status register. |
| 0x104 | SRW | sie | Supervisor interrupt enable register. |
| 0x105 | SRW | stvec | Supervisor trap handler base address. |
| 0x106 | SRW | scounter | Supervisor counter-enable register. |
| **Supervisor Trap Handling** | | | |
| 0x140 | SRW | sscratch | Scratch register for supervisor trap handlers. |
| 0x141 | SRW | sepc | Supervisor exception program counter. |
| 0x142 | SRW | scause | Supervisor trap cause. |
| 0x143 | SRW | stval | Supervisor bad address or instruction. |
| 0x144 | SRW | sip | Supervisor interrupt pending. |
| **Machine Information Registers** | | | |
| 0xF11 | MRO | mvendorid | Vendor ID. |
| 0xF12 | MRO | marchid | Architecture ID. |
| 0xF13 | MRO | mimpid | Implementation ID. |
| 0xF14 | MRO | mhartid | Hardware thread ID. |
| **Machine Trap Setup** | | | |
| 0x300 | MRW | mstatus | Machine status register. |
| 0x301 | MRO | misa | ISA and extensions. |
| 0x302 | MRW | medeleg | Machine exception delegation register. |
| 0x303 | MRW | mideleg | Machine interrupt delegation register. |
| 0x304 | MRW | mie | Machine interrupt enable register. |
| 0x305 | MRW | mtvec | Machine trap handler base address. |
| 0x306 | MRW | mcounteren | Machine counter-enable Register. |
| **Machine Trap Handling** | | | |
| 0x340 | MRW | mscratch | Scratch register for machine trap handlers. |
| 0x341 | MRW | mepc | Machine exception program counter. |

| Number | Privilege | Name | Description |
|---|---|---|---|
| 0x342 | MRO | mcause | Machine trap cause. |
| 0x343 | MRO | mtval | Machine bad address or instruction. |
| 0x344 | MRW | mip | Machine interrupt pending. |
| **Machine Counter/Timers** | | | |
| 0xB00 | MRW | mcycle | Machine cycle counter. |
| 0xB02 | MRW | minstret | Machine instructions-retired counter. |
| 0xB80 | MRW | mcycleh | Upper 32 bits of mcycle. |
| 0xB82 | MRW | minstreth | Upper 32 bits of minstret. |
| **Unprivileged Floating-Point CSRs** | | | |
| 0x001 | URW | fflags | Floating-point accrued exceptions. |
| 0x002 | URW | frm | Floating-point dynamic rounding mode. |
| 0x003 | URW | fcsr | Floating-point control and status register (frm + fflags). |

#### 2.2.1.11. Machine Trap-vector Base-address Register (mtvec) Vectored Mode

Version 2.2 of the RX processor core starts to support the vectored mode of the register mtvec and meets the demand of MODE field WARL (Write Any Values, Reads Legal Values) behavior, as shown in Figure 2.4.

| 31            2 | 1      0 | |
|---|---|---|
| BASE [31:2] (WARL) | MODE(WARL) | mtvec |
| 30 | 2 | |

**Figure 2.4. Machine Trap-vector Base-address Register (mtvec)**

According to the RISC-V Privileged Specification (Version 20211203), the mtvec mode decides the address where PC is to be set. You can change the vectored mode of the core by writing the MODE field of mtvec. The encoding of the MODE field is shown in Table 2.3.

**Table 2.3. mtvec Register**

<table>
<tr><th>Field</th><th>Name</th><th>Behavior</th><th>Width</th><th>Description</th></tr>
<tr><td>[31:2]</td><td>BASE</td><td>WARL</td><td>29</td><td>Vector base address.</td></tr>
<tr><td>[1:0]</td><td>MODE</td><td>WARL</td><td>3</td><td>Encoding of mtvec MODE field.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>Direct</td><td>All exceptions set PC to BASE.</td></tr>
<tr><td>1</td><td>Vectored</td><td>Asynchronous interrupts set PC to BASE+4×cause.</td></tr>
<tr><td>2</td><td>—</td><td>Reserved.</td></tr>
</table></td></tr>
</table>

When MODE=Direct, all traps into Machine mode cause the PC to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into Machine mode cause the PC to be set to the address in the BASE field, whereas interrupts cause the PC to be set to the address in the BASE field plus four times the interrupt cause number.

#### 2.2.1.12. PMP

The Physical Memory Protection (PMP) unit provides Machine mode control registers to limit the access of different regions of physical memory with different privileges, including read, write, and execute, for RV32 systems. To support Lattice RISC-V products, the PMP structure only supports the top boundary of an arbitrary range (TOR) mode with up to four entries and the granularity is 0. Our design follows the RISC-V Privileged Specification (Version 1.12).

PMP entries are described by an 8-bit configuration register and one 32-bit address register. These two kinds of registers are packed into CSRs to minimize context-switch time. The PMP configuration registers named pmpcfg#

determine the permission and addressing mode for protection regions. The PMP address registers named pmpaddr# contain the address for corresponding regions. # indicates the serial number of register.

This PMP unit partitions the memory range to four pages. There are only four entries for this unit instead of sixteen or sixty-four entries as in the RISC-V Specification. In other words, in this PMP unit, there is only one PMP configuration register, pmpcfg0, and four PMP address registers, pmpaddr0–pmpaddr3. All the registers fields are WARL registers.

- PMP Configuration Registers

Each pmpcfg# register contains four, 8-bits pmp#cfg register fields to describe the access privileges corresponding to four pmpaddr# for the RV32 system. As mentioned above, only pmpcfg0 is used in this unit and its associated number in CSRs is 0x3a0, as shown in Figure 2.5.

| 31 24 | 23 16 | 15 8 | 7 0 | |
|---|---|---|---|---|
| pmp3cfg | pmp2cfg | pmp1cfg | pmp0cfg | pmpcfg0 |
| 8 | 8 | 8 | 8 | |

**Figure 2.5. RV32 PMP Configuration CSR Layout**

Table 2.4 shows the layout of one pmp#cfg register inside pmpcfg0.

**Table 2.4. pmp#cfg Register Format**

<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Description</th></tr>
<tr><td>[7]</td><td>L</td><td>WARL</td><td>1</td><td>The PMP entry is locked.</td></tr>
<tr><td>[6:5]</td><td>0</td><td>WARL</td><td>2</td><td>—</td></tr>
<tr><td>[4:3]</td><td>A</td><td>WARL</td><td>2</td><td>Encoding the address-matching mode of the associate PMP address register.<table><tr><th>Value</th><th>Mode</th><th>Description</th></tr><tr><td>0</td><td>OFF</td><td>Null region (disabled)</td></tr><tr><td>1</td><td>TOR</td><td>Top of range</td></tr></table></td></tr>
<tr><td>[2]</td><td>X</td><td>WARL</td><td>2</td><td>When set, PMP entry permits instruction execution.<br>When clear, instruction execution is denied.</td></tr>
<tr><td>[1]</td><td>W</td><td>WARL</td><td>2</td><td>When set, PMP entry permits write.<br>When clear, write is denied.</td></tr>
<tr><td>[0]</td><td>R</td><td>WARL</td><td>2</td><td>When set, PMP entry permits read.<br>When clear, read is denied.</td></tr>
</table>

The R, W, and X bits determine whether this entry allows read, write, or execute respectively.

The A bits encode the address-matching mode. Unlike described in RISC-V Privileged Specification (Version 20211203), this field can only be in two modes, OFF or TOR. The NA4 and NAPOT modes are reserved for future requirements. The L bit indicates whether the entry is locked. When the L bit is set, writes to configuration register and related address registers are ignored. Locked PMP entries are unlocked when the hart is reset. For instance, if the entry *i* is locked, writes to pmpicfg and pmpaddri are ignored. Additionally, in TOR mode, writing to pmpaddri-1 is also ignored.

- PMP Address Registers

Each pmpaddr# indicates the bits [33:2] of a 34-bits physical address for RV32 systems, as shown in Figure 2.6. Four pmpaddr# are initialized in this unit and its associated number in CSRs are 0x3b0 to 0x3b3.

| 31 0 | |
|---|---|
| address [33:2] (WARL) | pmpaddr# |
| 32 | |

**Figure 2.6. PMP Address Register Format, RV32**

- Priority and Matching Logics

As shown in Table 2.5, this section describes the logic to verify the access to some region in physical memory. A PMP entry needs to fully match all bytes of an access and then the L, R, W, and X bits determine whether the access passes or fails. If L is clear and the privilege mode is M-Mode, the access succeeds. If L is set, or L is clear with the privilege mode in U-Mode or S-Mode, the access is determined by R, W, X bits. If no PMP entry matches an M-Mode access, the access succeeds. If no PMP entry matches an S-Mode or U-Mode access, but at least one entry is implemented, the access fails. If at least one access fails, an access-fault exception is generated. The L bit cannot be clear until system resets.

**Table 2.5. PMP Access Logic**

| Access Mode | Privilege Mode | Read | Write | Execute |
|---|---|---|---|---|
| Access in protected range | L = 0 & (M-Mode) | Succeeds | | |
| | L = 0 & (U-Mode \|\| S-Mode) | R bit | W bit | X bit |
| | L = 1 | R bit | W bit | X bit |
| Access not in protected range | M-Mode | Succeeds | | |
| | U-Mode \|\|S-Mode | Fails | | |
| Access cross protected and not protected range | Any Mode | Fails | | |
| All entries are off | M-Mode | Succeeds | | |
| All entries are off | U-Mode \|\| S-Mode | Fails | | |

#### 2.2.1.13. Configurable I/D Local Bus Interface Register Wrapper

Version 2.3 of the RX core allows you to configure if the I/D Local Bus interface register wrapper is enabled. You can configure this register wrapper enabling option in Lattice Propel Builder GUI to balance the RX core fmax and DMIPS. When enabling the register wrapper, fmax increases while DMIPS decreases. Conversely, when disabling the I/D register wrapper, fmax decreases while DMIPS increases.

#### 2.2.1.14. Write Response

Version 2.2 of the RX core starts to support the write response. This means a write error on the local and AXI bus on the processor causes the Store/AMO access fault exception of the core, exception ID: 7.

It makes sure write response is honored for both cache range and I/O range that have separate logic for dealing with write operations.

#### 2.2.1.15. CFU-LI

Custom Function Unit Logic Interface (CFU-LI) 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; and -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 RX 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 RX 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:

| 31 | 30 – 24 | 23 – 16 | 15 – 8 | 7 – 0 |
|---|---|---|---|---|
| en | reserved | state_id | reserved | cfu_id |

**Figure 2.7. 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.

| 31 – 25 | 24 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|---|
| cf_id[9:3] | rs2 | rs1 | cf_id[2:0] | rd | 0 0 0 1 0 1 1 |
| | | | | | custom-0 |

**Figure 2.8. 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.

| 31 – 24 | 23 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|---|
| imm[7:0] | cf_id[3:0] | rs1 | 0 0 0 | rd | 0 1 0 1 0 1 1 |
| | | | | | custom-1 |

**Figure 2.9.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.

| 31–25 | 24–20 | 19–15 | 14–12 | 11–7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cf_id[9:3] | rs2 | rs1 | cf_id[2:0] | custom | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| | | | | | custom-2 | | | | | | |

**Figure 2.10. CFU Flex-type Instruction Encoding**

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

| 31–7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| custom | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| | custom-2 | | | | | | |

**Figure 2.11. 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 32-7=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 2.12). 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.

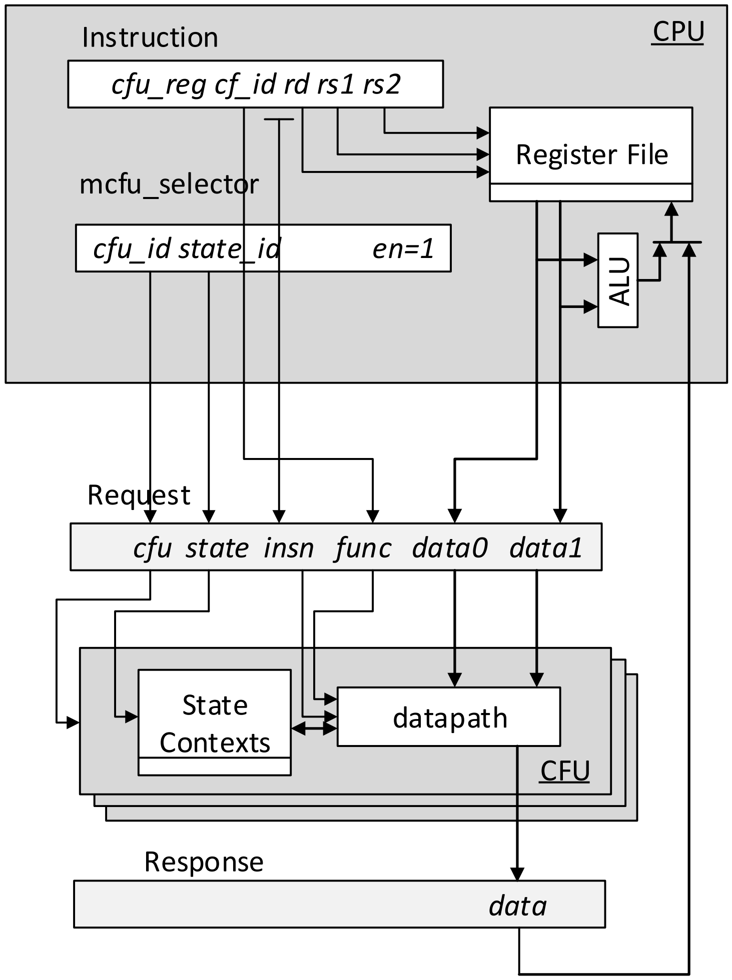


**Figure 2.12. Execution of a Custom Function Instruction**

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.

### 2.2.2. Submodule

All the submodules are covered here. Every submodule has a fixed base address. See Table 2.18.

#### 2.2.2.1. PLIC

The Platform Level Interrupt Controller (PLIC) module is compliant with the RISC-V Platform-Level Interrupt Controller Specification (Version 1.0).

The PLIC multiplexes various device interrupts onto the external interrupt lines of Hart contexts, with hardware support for interrupt priorities. Note: The context refers to the specific privilege mode in the specific Hart of specific RISC-V processor instance. PLIC supports up to 31 external interrupts (0 is reserved) with seven priority levels, and each one has a corresponding interrupt ID, starting from 1. The first input interrupt (#1) is fixed to Watchdog Timer device.

The PLIC has two interrupt output signals connected to external interrupt inputs of the CPU – one for Machine mode, the other for Supervisor mode.

Figure 2.13 shows the block diagram of PLIC operation parameter. An example for how it works: interrupt input 1 gets asserted, it goes through the Gateway and sets Interrupt Pending bit of the Source. If its Interrupt Enable (IE) is set, the priority value can be passed and compared to other inputs all the way through the chain. The interrupt ID is similarly forwarded. So, if the Max Priority is larger than the threshold, External Interrupt Pending (EIP) can be asserted and sent to the processor. Meanwhile, the Gateway blocks subsequent interrupts from being forwarded until the current interrupt has been completed. Target 0 goes to Machine mode external interrupt, and Target 1 goes to Supervisor mode external interrupt.

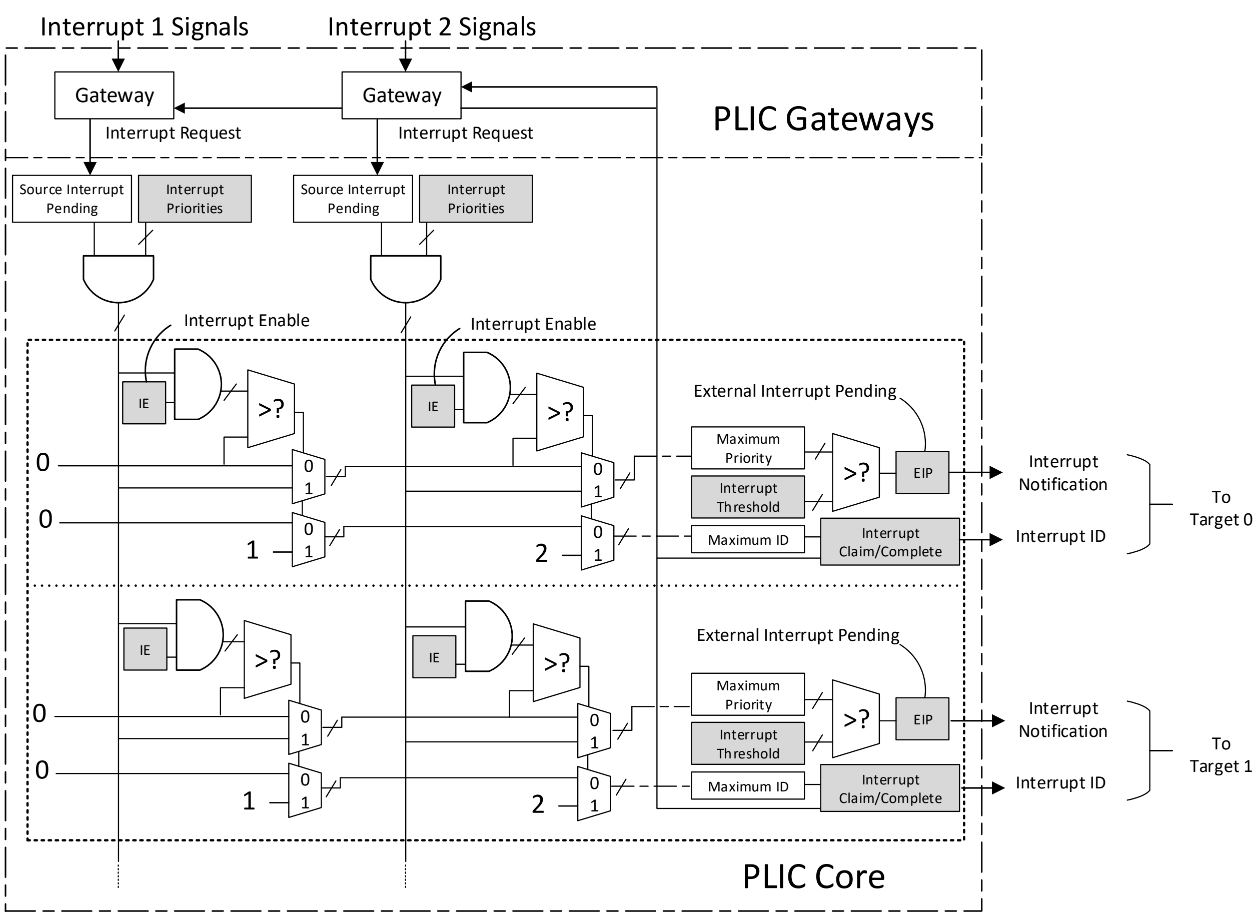


**Figure 2.13. PLIC Operation Parameter Block Diagram**

The following register blocks are defined in PLIC:

- Interrupt Priorities Registers

Each PLIC interrupt source can be assigned a priority by writing to its 32-bit memory-mapped priority register. A priority value of 0 is reserved to mean never interrupt and effectively disables the interrupt. Priority 1 is the lowest active priority while the maximum level of priority depends on user settings. For example, the highest priority is 3 if the width of PLIC Priority Register is set to two. Ties between global interrupts of the same priority are broken by the Interrupt ID. Interrupts with the lowest ID have the highest effective priority.

The base address of Interrupt Source Priority block within PLIC Memory Map region is fixed at 0x000000.

- Interrupt Pending Bits Registers

The current status of the interrupt source pending bits in the PLIC core can be read from the pending array, organized as 32-bit register. The pending bit for interrupt ID N is stored in bit N. Bit 0 of word 0, which represents the non-existent interrupt source 0, is hardwired to zero.

A pending bit in the PLIC core can be cleared by setting the associated enable bit then performing a claim.

The base address of Interrupt Pending Bits block within PLIC Memory Map region is fixed at 0x001000.

- Interrupt Enables Registers

Each global interrupt can be enabled by setting the corresponding bit in the enables registers. The enables registers are accessed as a contiguous array of 32-bit registers, packed the same way as the pending bits. Bit 0 of enable register 0 represents the non-existent interrupt ID 0 and is hardwired to 0. PLIC has two Interrupt Enable blocks, one for each context.

The context refers to the specific privilege mode in the specific Hart of specific RISC-V processor instance.

For the current IP, context 0 refers to hart 0 Machine mode and context 1 refers to hart 0 Supervisor mode.

The base address of Interrupt Enable Bits block within PLIC Memory Map region is fixed at 0x002000.

- Priority Thresholds Registers

PLIC provides context-based threshold register for the settings of an interrupt priority threshold of each context. The threshold register is a WARL field. The PLIC masks all PLIC interrupts of a priority less than or equal to threshold. For example, a threshold value of zero permits all interrupts with non-zero priority.

The base address of the Priority Thresholds Registers block is located at 4K alignment starting from offset 0x200000.

- Interrupt Claim Registers

The PLIC can perform an interrupt claim by reading the Claim/Complete Registers, which return the ID of the highest priority pending interrupt or zero if there is no pending interrupt. A successful claim also atomically clears the corresponding pending bit on the interrupt source.

The PLIC can perform a claim at any time and the claim operation is not affected by the setting of the Priority Thresholds Registers.

The Interrupt Claim Process Register is context-based and is located at 4K alignment + 4 starting from offset 0x200000.

- Interrupt Completion Registers

The PLIC signals the completion of executing an interrupt handler by host signaling the PLIC and writing the interrupt ID received from the claim to the Claim/Complete Register. The PLIC does not check whether or not the completion ID is the same as the last claim ID for that target. If the completion ID does not match an interrupt source that is currently enabled for the target, the completion is silently ignored.

The Interrupt Completion Registers are context-based and located at the same address as the Interrupt Claim Process Register, which is at 4K alignment + 4 starting from offset 0x200000.

Table 2.6 provides the description of PLIC registers.

**Table 2.6. PLIC Registers**

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x00_0000</td><td>—</td><td>Reserved<br>Interrupt source 0 does not exist.</td></tr>
<tr><td>0x00_0004</td><td>PLIC_PRIORITY_SRC1</td><td>Interrupt source 1 priority
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:3]</td><td>Reserved</td><td>RO</td><td>29</td><td>0x0</td></tr>
<tr><td>[2:0]</td><td>Priority</td><td>RW</td><td>3</td><td>0x0</td></tr>
</table>
Priority:<br>Sets the priority for a given global interrupt.</td></tr>
<tr><td>0x00_0008 ....<br>0x00_007C</td><td>PLIC_PRIORITY_SRC2<br>...<br>PLIC_PRIORITY_SRC31</td><td>Same as PLIC_PRIORITY_SRC1.</td></tr>
<tr><td>...</td><td>—</td><td>—</td></tr>
<tr><td>0x00_1000</td><td>PLIC_PENDING1</td><td>PLIC Interrupt Pending Register 1 (pending1)
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:1]</td><td>PendingN</td><td>RO</td><td>31</td><td>0x0</td></tr>
<tr><td>[0]</td><td>Pending0</td><td>RO</td><td>1</td><td>0x0</td></tr>
</table>
Pending0:<br>Non-existent global interrupt 0 is hardwired to zero.<br>PendingN:<br>Equal to PLIC_PENDING1[N], Pending bit for global interrupt N.</td></tr>
<tr><td>...</td><td>—</td><td>—</td></tr>
<tr><td>0x00_2000</td><td>PLIC_ENABLE1_M</td><td>PLIC Interrupt Enable Register 1 (enable1) for Hart 0 M-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:1]</td><td>EnableN</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>[0]</td><td>Enable0</td><td>RO</td><td>1</td><td>0x0</td></tr>
</table>
Enable0:<br>Non-existent global interrupt 0 is hardwired to zero.<br>EnableN:<br>Equal to PLIC_ENABLE_M[N], enable bit for global interrupt N.</td></tr>
<tr><td>...</td><td>—</td><td>—</td></tr>
<tr><td>0x00_2080</td><td>PLIC_ENABLE1_S</td><td>PLIC Interrupt Enable Register 1 (enable1) for Hart 0 S-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:1]</td><td>EnableN</td><td>RW</td><td>1</td><td>0x0</td></tr>
<tr><td>[0]</td><td>Enable0</td><td>RO</td><td>1</td><td>0x0</td></tr>
</table>
Enable0:<br>Non-existent global interrupt 0 is hardwired to zero.<br>EnableN:<br>Equal to PLIC_ENABLE_S[N], enable bit for global interrupt N.</td></tr>
<tr><td>...</td><td>—</td><td>—</td></tr>
</table>

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x20_0000</td><td>PLIC_THRESHOLD1_M</td><td>PLIC Interrupt Priority Threshold Register (threshold) for Hart 0 M-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:3]</td><td>Reserved</td><td>RO</td><td>29</td><td>0x0</td></tr>
<tr><td>[2:0]</td><td>Threshold</td><td>RW</td><td>3</td><td>0x0</td></tr>
</table>
Threshold:<br>Sets the priority threshold.</td></tr>
<tr><td>0x20_0004</td><td>PLIC_CLAIM_1_M</td><td>PLIC Claim Register (claim) for Hart 0 M-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>Claim</td><td>RO</td><td>32</td><td>0x0</td></tr>
</table>
Claim:<br>Read only field, which returns the ID of the highest priority pending interrupt or zero if there is no pending interrupt. A successful claim also atomically clears the corresponding pending bit on the interrupt source.</td></tr>
<tr><td>0x20_0004</td><td>PLIC_COMPLETE_1_M</td><td>PLIC Complete Register (complete) for Hart 0 M-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>Completion</td><td>WO</td><td>32</td><td>0x0</td></tr>
</table>
Completion:<br>Write only field, write to it to complete interrupt process.</td></tr>
<tr><td>…</td><td>—</td><td>—</td></tr>
<tr><td>0x20_1000</td><td>PLIC_THRESHOLD1_S</td><td>PLIC Interrupt Priority Threshold Register (threshold) for Hart 0 S-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:3]</td><td>Reserved</td><td>RO</td><td>29</td><td>0x0</td></tr>
<tr><td>[2:0]</td><td>Threshold</td><td>RW</td><td>3</td><td>0x0</td></tr>
</table>
Threshold:<br>Sets the priority threshold.</td></tr>
<tr><td>0x20_1004</td><td>PLIC_CLAIM_1_S</td><td>PLIC Claim Register (claim) for Hart 0 S-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>Claim</td><td>RO</td><td>32</td><td>0x0</td></tr>
</table>
Claim:<br>Read only field, which returns the ID of the highest priority pending interrupt or zero if there is no pending interrupt. A successful claim also atomically clears the corresponding pending bit on the interrupt source.</td></tr>
<tr><td>0x20_1004</td><td>PLIC_COMPLETE_1_S</td><td>PLIC Complete Register (complete) for Hart 0 S-Mode
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>Completion</td><td>WO</td><td>32</td><td>0x0</td></tr>
</table>
Completion:<br>Write only field, write to it to complete interrupt process.</td></tr>
</table>

#### 2.2.2.2. CLINT

The CLINT module implements mtime, mtimecmp, and some other memory-mapped CSR registers that are associated with timer and software interrupts.

There are two clocks for CLINT. The msip register is clocked by the system clock, while the mtimecmp and mtime are clocked by a real time clock which is typically 32 kHz for Lattice FPGA.

For Lattice Avant family devices, you cannot directly get the 32 kHz output from OSC and PLL. Thus, a 512 clock divider is designed for the real time clock port of the RX core, clk_realtime _i. It is recommended the input of the real time clock port be a 16.384 MHz clock signal, and it can later provide the 32 kHz clock signal to mtimecmp and mtime registers.

For other family devices, you can directly configure a 32 kHz output on OSC. Then, you can connect this low frequency clock to the real time clock port.

Table 2.7 provides the descriptions of CLINT registers.

**Table 2.7. CLINT Registers**

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x00_0000</td><td>CLINT_MSIP</td><td>MSIP Register for hart 0<br>
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:1]</td><td>Reserved</td><td>RO</td><td>31</td><td>0x0</td></tr>
<tr><td>[0]</td><td>msip</td><td>RW</td><td>1</td><td>0x0</td></tr>
</table>
msip:<br>Reflects the memory-mapped MSIP bit of the mip CSR Register.<br>Writing a 1 in msip field results in the generation of software interrupt.</td></tr>
<tr><td>…</td><td>—</td><td>—</td></tr>
<tr><td>0x00_4000</td><td>CLINT_MTIMECMP_L</td><td>Machine Timer Register – mtimecmp<br>
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>mtimecmp_l</td><td>RW</td><td>32</td><td>Unchanged</td></tr>
</table>
mtimecmp_l:<br>Lower 32 bits of mtimecmp CSR Register. The first reset value is 0xFFFF_FFFF, after first write, the reset does not change the value of this field.</td></tr>
<tr><td>0x00_4004</td><td>CLINT_MTIMECMP_H</td><td>Machine Timer Register – mtimecmp<br>
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>mtimecmp_h</td><td>RW</td><td>32</td><td>Unchanged</td></tr>
</table>
mtimecmp_h:<br>Higher 32 bits of mtimecmp CSR Register. The first reset value is 0xFFFF_FFFF, after first write, the reset does not change the value of this field.</td></tr>
<tr><td>…</td><td>—</td><td>—</td></tr>
<tr><td>0x00_BFF8</td><td>CLINT_MTIME_L</td><td>Machine Timer Register - mtime<br>
<table>
<tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr>
<tr><td>[31:0]</td><td>mtime_l</td><td>RW</td><td>32</td><td>0x0</td></tr>
</table>
mtime_l:<br>Lower 32 bits of mtime CSR Register.</td></tr>
</table>

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x00_BFFC</td><td>CLINT_MTIME_H</td><td>Machine Timer Register - mtime<br><table><tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr><tr><td>[31:0]</td><td>mtime_h</td><td>RW</td><td>32</td><td>0x0</td></tr></table><br>mtime_h:<br>Higher 32 bits of mtime CSR Register.</td></tr>
</table>

#### 2.2.2.3. Watchdog Timer

The watchdog timer device (WDT) provides a simple two-stage timer controlled through one memory-mapped CSR register, WDCSR.

WDT waits a software-configured period of time with the expectation that system software re-initializes the watchdog state within this period of time. If this time period elapses without software re-init occurring, then a first-stage timeout register bit S1WTO is set within WDCSR that asserts an interrupt request output signal to notify the system of a stage 1 watchdog timeout. If a second period of time elapses without software re-init of the watchdog, then a second-stage timeout register bit S2WTO is set within WDCSR that generates a separate interrupt request output signal to notify the system of a stage 2 watchdog timeout.

For current IP, the stage 1 watchdog timeout is connected to PLIC input channel 1 (fixed) and stage 2 watchdog timeout is connected to system reset.

The mtime CSR Register provides the time base for the watchdog timeout period. The timeout period itself – in units of watchdog clock tick – is specified by the WTOCNT field of the WDCSR CSR Register. When WDCSR is written, the WTOCNT value initializes a down counter that decrements with each watchdog tick.

The watchdog tick occurs when bit 14 of mtime transitions from 0 to 1. So the watchdog timeout period is 0.512 second, based on real time clock of 32 kHz. Meanwhile, the maximum timeout period (WTOCNT = 0x3FF) is about 524 seconds.

WDT is included in the CLINT module. WDT shares the same base address (0xF200_0000) with CLINT. Table 2.8 provides the description of WDT registers.

**Table 2.8. WDT Registers**

<table>
<tr><th>Offset</th><th>Name</th><th>Description</th></tr>
<tr><td>0x00_D000</td><td>WDT_WDCSR</td><td>Watchdog Register<br><table><tr><th>Field</th><th>Name</th><th>Access</th><th>Width</th><th>Reset</th></tr><tr><td>[31:14]</td><td>Reserved</td><td>RO</td><td>18</td><td>0x0</td></tr><tr><td>[13:4]</td><td>WTOCNT</td><td>RW</td><td>10</td><td>0x0</td></tr><tr><td>[3]</td><td>S2WTO</td><td>RW</td><td>1</td><td>0x0</td></tr><tr><td>[2]</td><td>S1WTO</td><td>RW</td><td>1</td><td>0x0</td></tr><tr><td>[1]</td><td>Reserved</td><td>RW</td><td>1</td><td>0x0</td></tr><tr><td>[0]</td><td>WDEN</td><td>RW</td><td>1</td><td>0x0</td></tr></table><br>WDEN:<br>When set, enables the WDT. When clear, the WDT is disabled and S1WTO and S2WTO output signals are forced to be zero (de-asserted). When system reset is asserted, WDT is disabled accordingly by setting WDEN to 0.<br>S1WTO:<br>Stage 1 watchdog timeout, active high.<br>S2WTO:<br>Stage 2 watchdog timeout, active high.<br>WTOCNT:<br>10-bit timeout counter. If it is non-zero and WDEN is set, it decrements every timeout period.</td></tr>
</table>

#### 2.2.2.4. UART

There is an optional local UART. This UART has fixed memory assignment.

## 2.3. Signal Description

Table 2.9 to Table 2.15 list the ports of the RX CPU soft IP in different categories.

### 2.3.1. sysClock and Reset

**Table 2.9. Clock and Reset Ports**

| Name | Direction | Width | Description |
|---|---|---|---|
| clk_system_i | In | 1 | High speed system clock input. |
| clk_realtime_i | In | 1 | Low speed real time clock input. |
| rstn_i | In | 1 | System reset, active low. |
| system_resetn_o | Out | 1 | Combined system reset and Debug Reset from JTAG. |
| uart_rxd_i | In | 1 | Input rxd for local UART. Only available when UART_EN enabled. |
| uart_txd_o | Out | 1 | Output txd for local UART. Only available when UART_EN enabled. |

### 2.3.2. Data Interface

The RX first release includes a local TCM and it is connected to the instruction port of the RISC-V core directly. So, there is no exported instruction interface. In certain scenarios, there is a need to have an exported instruction port. For example, the instruction may come from an external flash through a flash controller.

To support this, we provide an optional local bus port to connect TCM, while an extra optional AXI Interface for instruction port is available for accessing other memory mapped components such as a flash controller or DDR controller.

So, there are options for you to enable either Local Bus Interface for TCM or AXI Interface for external memory, or both Local Bus and AXI Interface.

Meanwhile, to remove potential dependency on other components at SoC level, there is an option for you to enable register slice for AXI-based instruction or data port, as shown in Figure 2.1.

Version 2.2 of the RX core starts to add CFU-LI support for you to connect a CFU accelerator.

**Table 2.10. 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 |

**Table 2.11. Local Data Ports (Optional)**

| Name | Direction | Width | Group | Description |
|---|---|---|---|---|
| LOCAL_BUS_M_DATA_cmd_valid | Out | 1 | Local Bus Command | — |
| LOCAL_BUS_M_DATA_cmd_ready | In | 1 | | — |
| LOCAL_BUS_M_DATA_cmd_payload_wr | Out | 1 | | — |
| LOCAL_BUS_M_DATA_cmd_payload_uncached | Out | 1 | | Fixed 1'b0. |
| LOCAL_BUS_M_DATA_cmd_payload_address | Out | 32 | | — |
| LOCAL_BUS_M_DATA_cmd_payload_data | Out | 32 | | — |
| LOCAL_BUS_M_DATA_cmd_payload_mask | Out | 4 | | The width field of load or store instruction. |
| LOCAL_BUS_M_DATA_cmd_payload_size | Out | 3 | | 3'b101: an 8-words read burst transfer.<br>3'b010: a single burst transfer. |
| LOCAL_BUS_M_DATA_cmd_payload_last | Out | 1 | | — |
| LOCAL_BUS_M_DATA_rsp_valid | In | 1 | Local Bus Read Response | — |
| LOCAL_BUS_M_DATA_rsp_payload_last | In | 1 | | — |
| LOCAL_BUS_M_DATA_rsp_payload_data[31:0] | In | 32 | | — |
| LOCAL_BUS_M_DATA_rsp_payload_error | In | 1 | | — |

**Table 2.12. Local Instruction Ports (Optional)**

| Name | Direction | Width | Group | Description |
|---|---|---|---|---|
| LOCAL_BUS_M_INSTR_cmd_valid | Out | 1 | Local Bus Command | — |
| LOCAL_BUS_M_INSTR_cmd_ready | In | 1 | | — |
| LOCAL_BUS_M_INSTR_cmd_payload_wr | Out | 1 | | Fixed 1'b0. |
| LOCAL_BUS_M_INSTR_ cmd_payload_uncached | Out | 1 | | Fixed 1'b0. |
| LOCAL_BUS_M_INSTR_cmd_payload_address | Out | 32 | | — |
| LOCAL_BUS_M_INSTR_cmd_payload_data | Out | 32 | | Fixed 32'b0. |
| LOCAL_BUS_M_INSTR_cmd_payload_mask | Out | 4 | | Fixed 4'b0. |
| LOCAL_BUS_M_INSTR_cmd_payload_size | Out | 3 | | 3'b101: an 8-words read burst transfer.<br>3'b010: a single burst transfer. |
| LOCAL_BUS_M_INSTR_cmd_payload_last | Out | 1 | | Fixed 4'b0. |
| LOCAL_BUS_M_INSTR_rsp_valid | In | 1 | Local Bus Read Response | — |
| LOCAL_BUS_M_INSTR_rsp_payload_last | In | 1 | | — |
| LOCAL_BUS_M_INSTR_rsp_payload_data[31:0] | In | 32 | | — |
| LOCAL_BUS_M_INSTR_rsp_payload_error | In | 1 | | — |

**Table 2.13. AXI Data Ports (Fixed)[1]**

| Name | Direction | Width | Group | Description |
|---|---|---|---|---|
| AXI_M_DATA_AWREADY | In | 1 | AXI4 Manager Write Address Channel | — |
| AXI_M_DATA_AWVALID | Out | 1 | | — |
| AXI_M_DATA_AWADDR | Out | 32 | | — |
| AXI_M_DATA_AWLEN | Out | 8 | | — |
| AXI_M_DATA_AWSIZE | Out | 3 | | — |
| AXI_M_DATA_AWBURST | Out | 2 | | Not implemented. |
| AXI_M_DATA_AWLOCK | Out | 1 | | Not implemented. |
| AXI_M_DATA_AWCACHE | Out | 4 | | Not implemented. |
| AXI_M_DATA_AWPROT | Out | 3 | | Not implemented. |
| AXI_M_DATA_AWQOS | Out | 4 | | Not implemented. |
| AXI_M_DATA_AWREGION | Out | 4 | | Not implemented. |
| AXI_M_DATA_AWID | Out | 1 | | Not implemented. |
| AXI_M_DATA_WREADY | In | 1 | AXI4 Manager Write Data Channel | — |
| AXI_M_DATA_WVALID | Out | 1 | | — |
| AXI_M_DATA_WDATA | Out | 32 | | — |
| AXI_M_DATA_WLAST | Out | 1 | | Not implemented. |
| AXI_M_DATA_WSTRB | Out | 4 | | — |
| AXI_M_DATA_BVALID | In | 1 | AXI4 Manager Write Response Channel | — |
| AXI_M_DATA_BRESP | In | 2 | | b'00: OKAY.<br>b'10: SLVERR.<br>b'11: DECERR. |
| AXI_M_DATA_BID | In | 1 | | Not implemented. |
| AXI_M_DATA_BREADY | Out | 1 | | — |
| AXI_M_DATA_ARVALID | In | 1 | AXI4 Manager Read Address Channel | — |
| AXI_M_DATA_ARREADY | Out | 1 | | — |
| AXI_M_DATA_ARCACHE | Out | 4 | | Not implemented. |
| AXI_M_DATA_ARPROT | Out | 3 | | Not implemented. |
| AXI_M_DATA_ARQOS | Out | 4 | | Not implemented. |
| AXI_M_DATA_ARREGION | Out | 4 | | Not implemented. |
| AXI_M_DATA_ARID | Out | 1 | | Not implemented. |
| AXI_M_DATA_ARADDR | Out | 32 | | — |
| AXI_M_DATA_ARLEN | Out | 8 | | — |
| AXI_M_DATA_ARSIZE | Out | 3 | | — |
| AXI_M_DATA_ARBURST | Out | 2 | | Fixed 2'b01. |
| AXI_M_DATA_ARLOCK | Out | 1 | | Not implemented. |
| AXI_M_DATA_RID | In | 1 | AXI4 Manager Read Data Channel | Not implemented. |
| AXI_M_DATA_RDATA | In | 32 | | — |
| AXI_M_DATA_RRESP | In | 2 | | b'00: OKAY.<br>b'10: SLVERR.<br>b'11: DECERR. |
| AXI_M_DATA_RLAST | In | 1 | | — |
| AXI_M_DATA_RVALID | In | 1 | | — |
| AXI_M_DATA_RREADY | Out | 1 | | — |

**Note:**

1. Optional interfaces can be configured through the RX IP block wizard GUI upon your need. Meanwhile, the fixed interface is necessary for the RX core and cannot be configured through the GUI.

**Table 2.14. AXI Instruction Ports (Optional)**

| Name | Direction | Width | Group | Description |
|---|---|---|---|---|
| AXI_M_INSTR_AWREADY | In | 1 | AXI4 Manager Write Address Channel | Not used. |
| AXI_M_INSTR_AWVALID | Out | 1 | | Not used. |
| AXI_M_INSTR_AWADDR | Out | 32 | | Not used. |
| AXI_M_INSTR_AWLEN | Out | 8 | | Not used. |
| AXI_M_INSTR_AWSIZE | Out | 3 | | Not used. |
| AXI_M_INSTR_AWBURST | Out | 2 | | Not used. |
| AXI_M_INSTR_AWLOCK | Out | 1 | | Not used. |
| AXI_M_INSTR_AWCACHE | Out | 4 | | Not used. |
| AXI_M_INSTR_AWPROT | Out | 3 | | Not used. |
| AXI_M_INSTR_AWQOS | Out | 4 | | Not used. |
| AXI_M_INSTR_AWREGION | Out | 4 | | Not used. |
| AXI_M_INSTR_AWID | Out | 1 | | Not used. |
| AXI_M_INSTR_WREADY | In | 1 | AXI4 Manager Write Data Channel | Not used. |
| AXI_M_INSTR_WVALID | Out | 1 | | Not used. |
| AXI_M_INSTR_WDATA | Out | 32 | | Not used. |
| AXI_M_INSTR_WLAST | Out | 1 | | Not used. |
| AXI_M_INSTR_WSTRB | Out | 4 | | Not used. |
| AXI_M_INSTR_BVALID | In | 1 | AXI4 Manager Write Response Channel | Not used. |
| AXI_M_INSTR_BRESP | In | 2 | | Not used. |
| AXI_M_INSTR_BID | In | 1 | | Not used. |
| AXI_M_INSTR_BREADY | Out | 1 | | Not used. |
| AXI_M_INSTR_ARVALID | In | 1 | AXI4 Manager Read Address Channel | — |
| AXI_M_INSTR_ARREADY | Out | 1 | | — |
| AXI_M_INSTR_ARCACHE | Out | 4 | | Not implemented. |
| AXI_M_INSTR_ARPROT | Out | 3 | | Not implemented. |
| AXI_M_INSTR_ARQOS | Out | 4 | | Not implemented. |
| AXI_M_INSTR_ARREGION | Out | 4 | | Not implemented. |
| AXI_M_INSTR_ARID | Out | 1 | | Not implemented. |
| AXI_M_INSTR_ARADDR | Out | 32 | | — |
| AXI_M_INSTR_ARLEN | Out | 8 | | — |
| AXI_M_INSTR_ARSIZE | Out | 3 | | Fixed 2'b10. |
| AXI_M_INSTR_ARBURST | Out | 2 | | Fixed 2'b01. |
| AXI_M_INSTR_ARLOCK | Out | 1 | | Not implemented. |
| AXI_M_INSTR_RID | In | 1 | AXI4 Manager Read Data Channel | Not implemented. |
| AXI_M_INSTR_RDATA | In | 32 | | — |
| AXI_M_INSTR_RRESP | In | 2 | | b’00: OKAY.<br>b’10: SLVERR.<br>b’11: DECERR. |
| AXI_M_INSTR_RLAST | In | 1 | | — |
| AXI_M_INSTR_RVALID | In | 1 | | — |
| AXI_M_INSTR_RREADY | Out | 1 | | — |

### 2.3.3. Interrupt Interface

**Table 2.15. Interrupt Ports**

| Name | Type | Width | Description |
|---|---|---|---|
| EXT_IRQ_Sx | In | 2 ~ 14 | Peripheral interrupts. |

## 2.4. Attribute Summary

The configurable attributes are shown in Table 2.16 and are described in Table 2.17.

The attributes can be configured through the Propel Builder software.

**Table 2.16. Configurable Attributes**

| Attribute | Selectable Values | Default | Dependency on Other Attributes/Device Family |
|---|---|---|---|
| **General** | | | |
| Processor Mode | Balanced | Balanced | — |
| **Debug configuration** | | | |
| Debug Enable | Disabled, Enabled | Enabled | — |
| Soft JTAG Enable | Disabled, Enabled | Enabled (for Lattice Avant family devices)<br>Disabled (for other family devices) | Debug Enable |
| JTAG Channel Selection | 14, 15, 16 | 14 | Debug Enable |
| **TCM Local Bus Ports Configuration[1]** | | | |
| TCM Enable | Disabled, Enabled (with AXI Instruction Ports Enabled)<br>Enabled (with AXI Instruction Ports Disabled) | Enabled | AXI Instruction Ports Enable |
| **AXI Instruction Ports Configuration[1]** | | | |
| AXI Instruction Ports Enable | Disabled, Enabled (with TCM Enabled)<br>Enabled (with TCM Disabled) | Enabled | TCM Enable |
| AXI Register Slice Type | 0, 1, 2 | 0 | AXI Instruction Ports Enable |
| **CFU Configuration** | | | |
| Enable CFU Ports | Disabled, Enabled | Disabled | — |
| Number of CFU | 1, 2 | 1 | Enable CFU Ports |
| **PLIC Configuration** | | | |
| Number of User Interrupt Requests | 2 ~ 14 | 8 | — |
| Interrupt for Supervisor Mode | Disabled, Enabled | Disabled | — |
| Width of PLIC priority register | 2, 3 | 3 | — |
| **NMI Configuration** | | | |
| Non-maskable interrupt enable | Disabled, Enabled | Disabled | — |
| **Local UART** | | | |
| Enable UART instance | Disabled, Enabled | Disabled | — |
| System Clock Frequency (MHz) | 2 - 200 | 100 | Enable UART Instance |
| Serial Data Width | 5, 6, 7, 8 | 8 | Enable UART Instance |

| Attribute | Selectable Values | Default | Dependency on Other Attributes/Device Family |
|---|---|---|---|
| Stop Bits | 0, 1 | 1 | Enable UART Instance |
| Parity Enable | Disabled, Enabled | Disabled | Enable UART Instance |
| UART Standard Baud Rate | 2400, 4800, 9600, 14400, 19200, 28800, 38400, 56000, 57600, 115200 | 115200 | Enable UART Instance |

**Note:**

1. Either the AXI Instruction Port or both of the TCM ports must be enabled to make the RX core perform normally.

**Table 2.17. Attributes Description**

| Attribute | Description |
|---|---|
| **General** | |
| Processor Mode | Balanced Mode offers RV32IMC with some other feature sets not accessible to users to get the most comprehensive performance. |
| **Debug configuration** | |
| Debug Enable | Whether to enable Debug module or not. |
| Soft JTAG Enable | Whether to enable Soft JTAG or not. |
| JTAG Channel Selection | Select the channel of harden JTAG block. |
| **TCM Local Bus Ports Configuration** | |
| TCM Enable | Whether to enable TCM Local Bus Ports or not. |
| **AXI Instruction Ports Configuration** | |
| AXI Instruction Ports enable | Whether to enable AXI Instruction Ports or not. |
| AXI Register Slice Type | Type of AXI Instruction Ports channel Register Slice.<br>0 – Bypass register slice<br>1 – Simple buffer<br>2 – Skid buffer |
| **CFU Configuration** | |
| Enable CFU Ports | Whether to enable CFU Ports or not. |
| Number of CFU | Number of CFU Ports. |
| **PLIC Configuration** | |
| Number of User Interrupt Requests | Number of Interrupt for peripherals. |
| Interrupt for Supervisor Mode | Whether or not to enable interrupt for Supervisor mode. If not, then all external interrupts go to Machine mode only. |
| Width of PLIC priority register | Data width of PLIC priority register. Default to 3 bits, so 8 priority levels in total. |
| Non-maskable interrupt enable | Configure the reset mode to be either sync or async. |
| **Local UART** | |
| Enable UART instance | Disabled, Enabled |
| System Clock Frequency (MHz) | Specifies the target frequency of the system clock. This is used for baud rate calculation. |
| Serial Data Width | Specifies the default data bit width of UART transactions. |
| Stop Bits | Specifies the default number of stop bits to be transmitted and received. |
| Parity Enable | Specifies the absence/presence of parity. |
| UART Standard Baud Rate | Selects between Standard Baud Rate and Custom Baud Rate for the reset value of Divisor Latch Register. The selected baud rate is used to set the reset value of Divisor Latch Register as follows: {DLR_MSB, DLR_LSB} = System Clock Frequency (MHz) x 1000000 / Selected Baud Rate. |

## 2.5. Memory Map

To achieve better overall performance, this IP separates the whole 4G memory range into several sections with some usage convention (Table 2.18). The region #0 is mandatory because it is a cacheable range. Other regions are optional.

**Table 2.18. SoC Memory Map**

| Base Address | Range | End Address | Description |
|---|---|---|---|
| **Region #0 (0x0000_0000 - 0x0FFF_FFFF) - RISC-V RX IP** | | | |
| 0x0000_0000 | 128KB | 0x0001_FFFF | TCM (enable TCM)/ User Memory extension (disable TCM). |
| 0x0002_0000 | —[1] | 0x0FFF_FFFF | User Memory extension. |
| **Region #15 (0xF000_0000 - 0xFFFF_FFFF) - RISC-V RX IP** | | | |
| 0xF000_0000 | 1KB | 0xF000_03FF | Local UART when UART_EN asserted; otherwise, reserved. |
| 0xF000_0400 | 32767KB | 0xF1FF_FFFF | Reserved. |
| 0xF200_0000 | 1024KB | 0xF20F_FFFF | CLINT & Watchdog Timer. |
| 0xF210_0000 | NA | 0xFBFF_FFFF | Reserved. |
| 0xFC00_0000 | 4096KB | 0xFC3F_FFFF | PLIC. |
| 0xFC40_0000 | NA | 0xFFFF_FFFF | Reserved. |
| **Region #1 (0x1000_0000 - 0x1FFF_FFFF)** | | | |
| … | —[1] | —[1] | —[1] |
| **Region #2 (0x2000_0000 - 0x2FFF_FFFF)** | | | |
| … | —[1] | —[1] | —[1] |

**Note**:

1. The actual valid base address/range/end address is determined by the user SoC design.

The total 4G memory space is divided into 16 256 MB regions to ease potential future PMP settings.

Processor cache range is 0x0000_0000 to 0x0FFF_FFFF, so region #0 is the only region for cacheable components. The first 128 KB (0x0000_0000 to 0x0001_FFFF) are reserved for TCM. The remaining spaces are for user external memory extension, either on-chip EBR-based memory or off-chip memory like flash and SDRAM.

Region #15 is reserved for RISC-V RX IP – local UART, CLINT, Watchdog Timer, and PLIC are assigned to this region.

All the other regions are for user extension.

# 3. RISC-V RX CPU IP Generation

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

To generate the RX IP Core module:

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

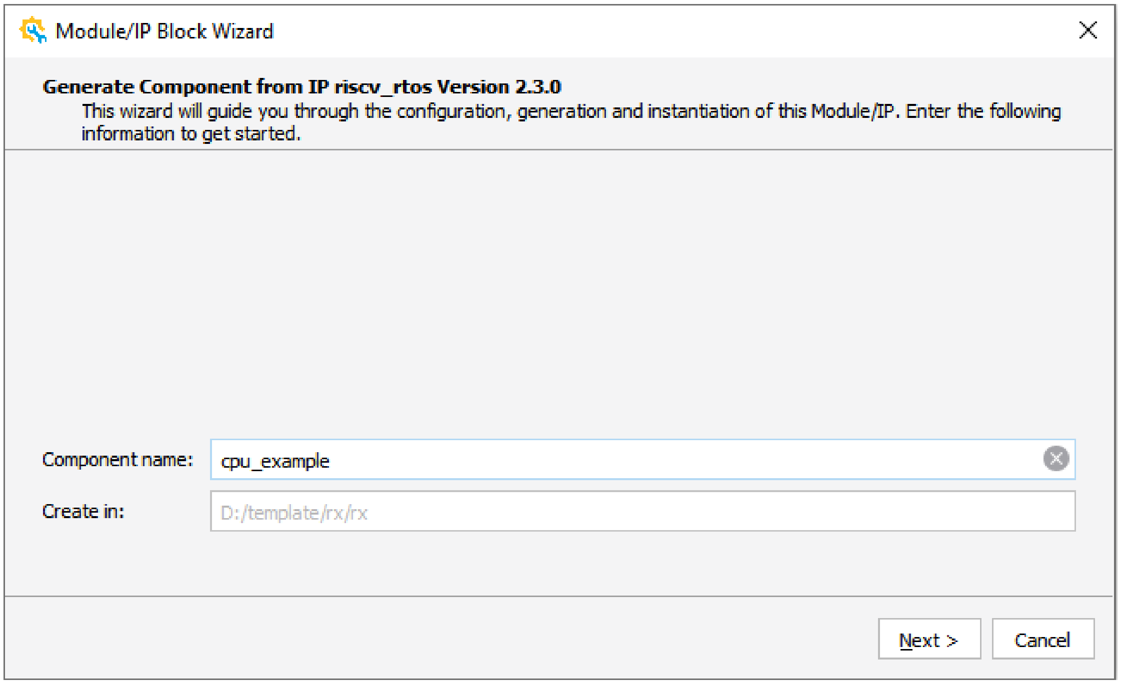


**Figure 3.1. Entering Component Name**

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

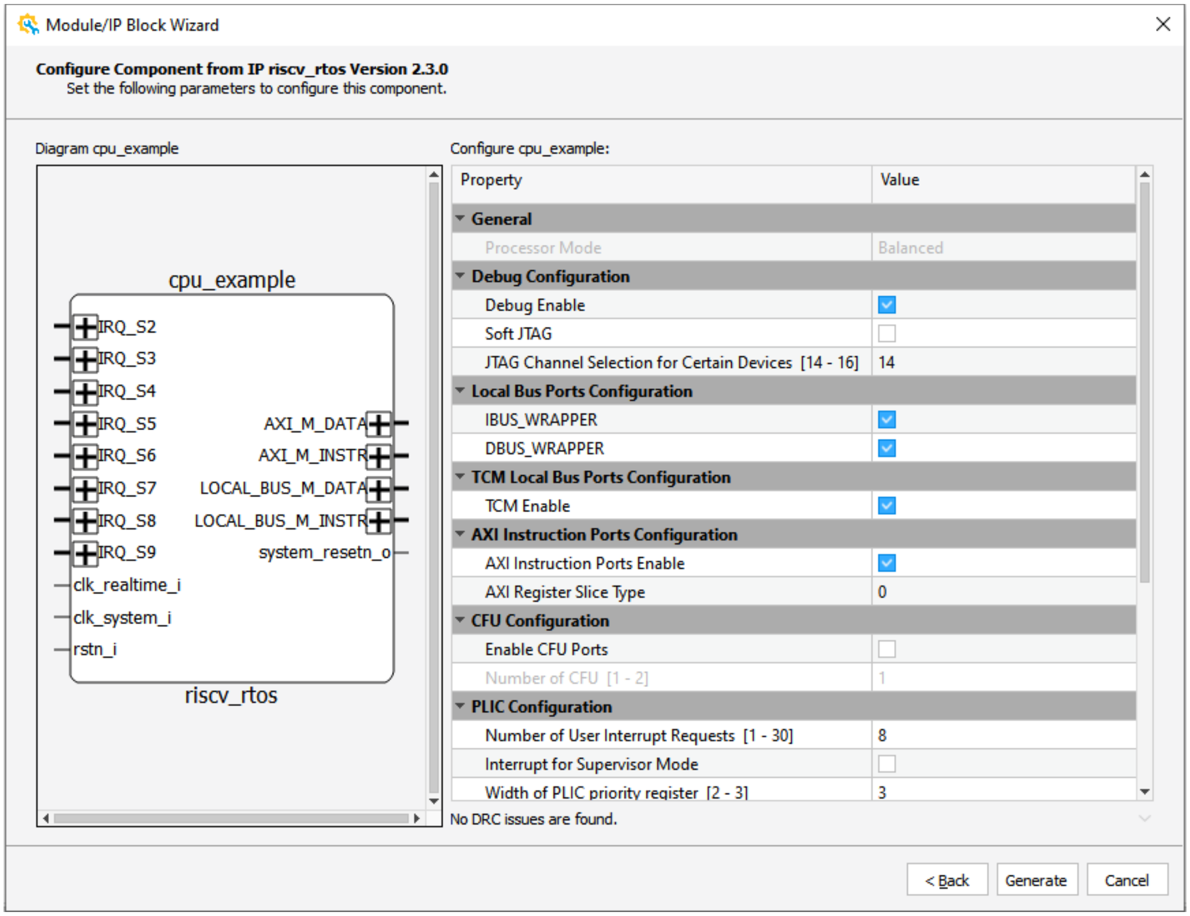


**Figure 3.2. Configuring Parameters**

4. Verify the information, as shown in Figure 3.3. Click **Finish**.

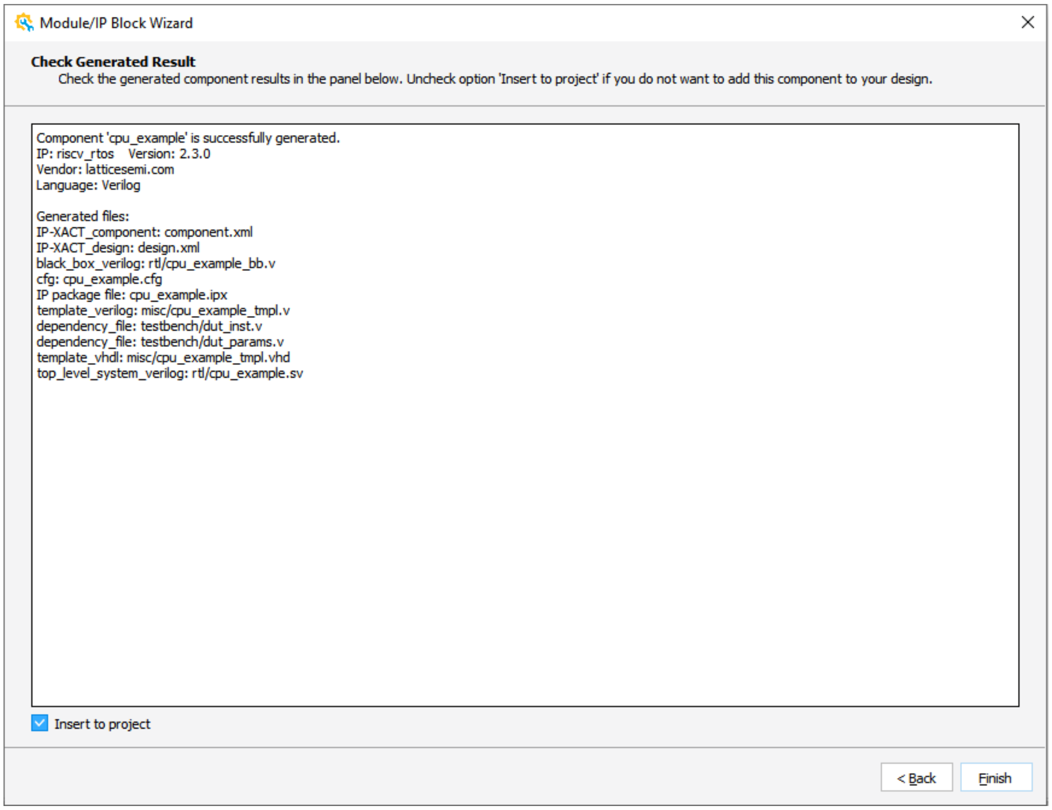


**Figure 3.3. Verifying Results**

5. Confirm or modify the module instance name, as shown in Figure 3.4. Click **OK**.

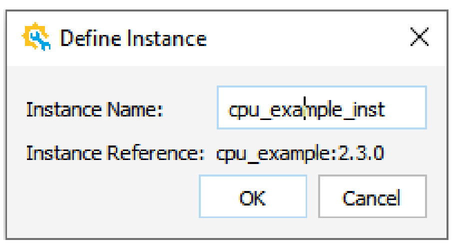


**Figure 3.4. Specifying Instance Name**

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

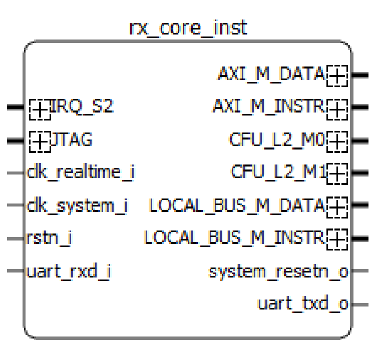


**Figure 3.5. Generated Instance**

# Appendix A. Resource Utilization

**Table A.1. Resource Utilization in CertusPro-NX Device**

| Configuration | LUTs | Registers | sysMEM EBRs |
|---|---|---|---|
| Processor core | 8846 | 4813 | 21 |
| Processor core + PLIC + CLINT + UART + CFU-LI + Debug | 9827 | 5718 | 21 |

**Note:** Resource utilization characteristics are generated using Lattice Radiant 2022.1 software.

**Table A.2. Resource Utilization in Lattice Avant Device**

| Configuration | LUTs | Registers | sysMEM EBRs |
|---|---|---|---|
| Processor core | 8673 | 4537 | 20 |
| Processor core + PLIC + CLINT + UART + CFU-LI + Debug | 9570 | 5441 | 20 |

**Note:** Resource utilization characteristics are generated using Lattice Radiant 2022.1 software.

# Appendix B. Debug with Soft JTAG

To debug with Soft JTAG:

1. In Propel Builder, enable **Soft JTAG** in the 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).

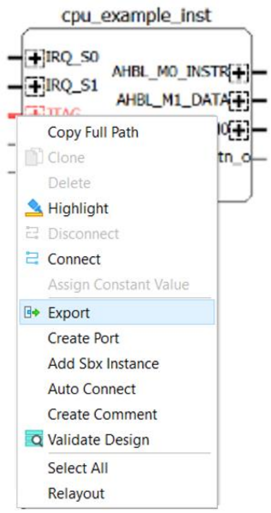


**Figure B.1. Exporting Pins**

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).

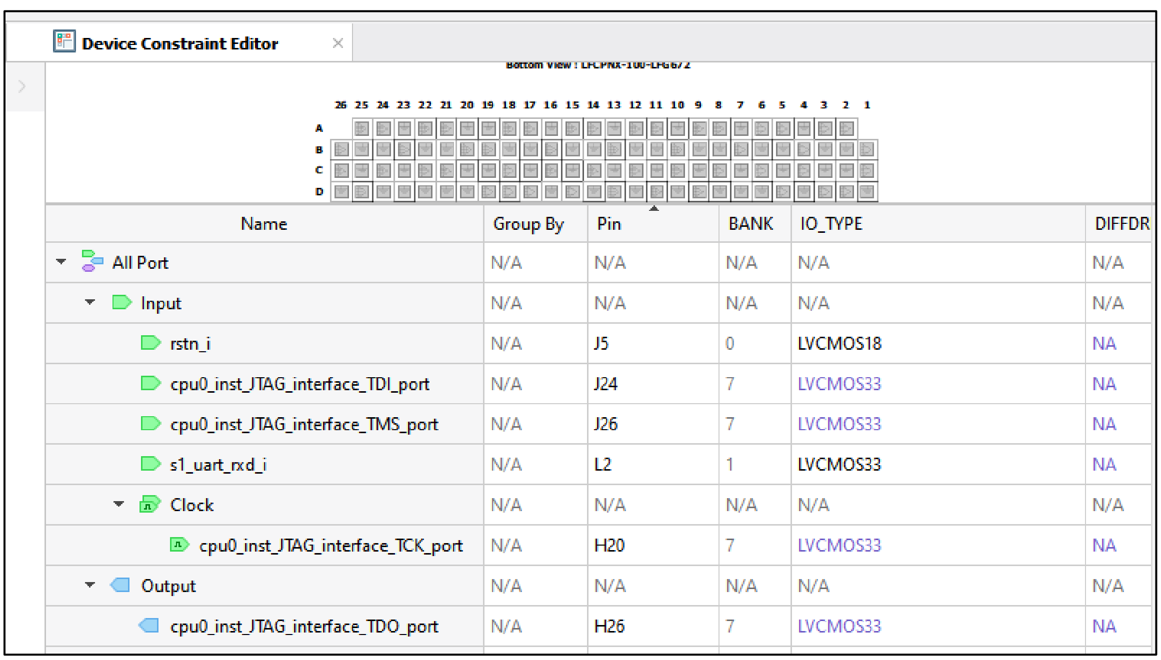


**Figure B.2. Assigning Pins**

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" to the Value of **Command Line Options** if TCK connects to normal I/O (Figure B.3).

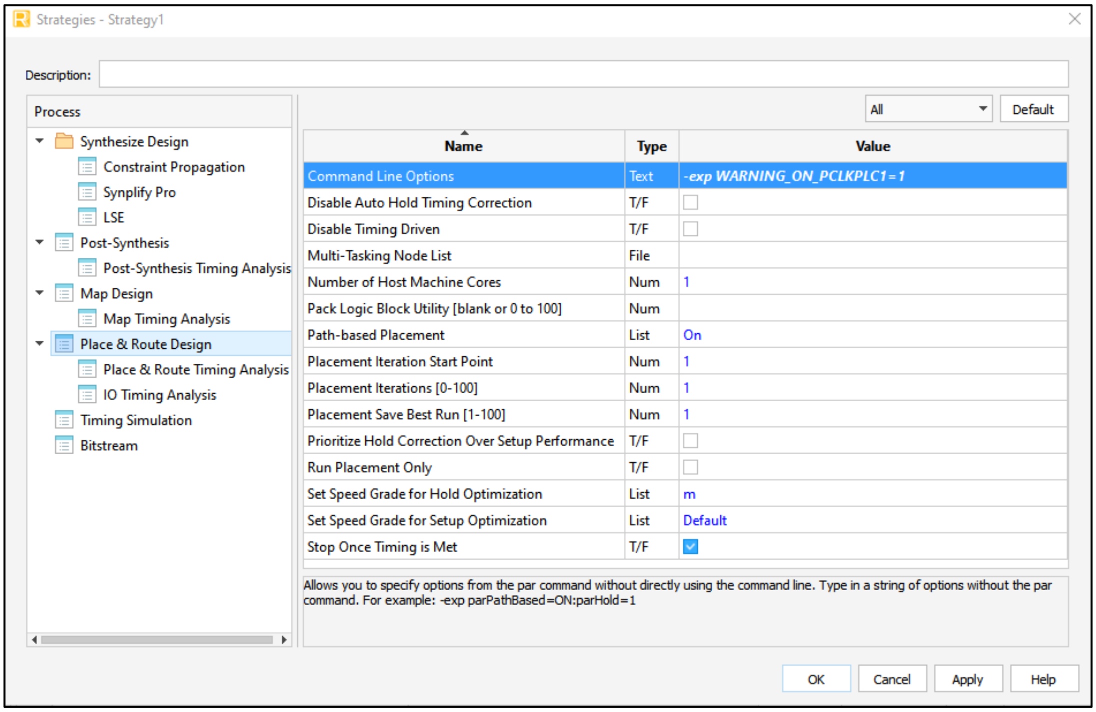

**Figure B.3. Setting Environment Variables**

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 Propel SDK 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

- RISC-V Composable Custom Extensions Specification (Draft)
- RISC-V Instruction Set Manual Volume I: Unprivileged ISA (20191213)
- RISC-V Instruction Set Manual Volume II: Privileged Architecture (20211203)
- RISC-V Privileged Specification Version 1.12
- RISC-V Platform Specification Version 0.2
- RISC-V Platform-Level Interrupt Controller Specification Version 1.0
- SiFive Interrupt Cookbook v1.2
- RISC-V Watchdog Timer Specification Version 1.0-draft-0.5
- AMBA 3 AHB-Lite Protocol v1.0
- AMBA AXI and ACE Protocol Specification vF.b
- Local Bus Specification
- Lattice Propel Builder 2023.2 User Guide (FPGA-UG-02196)

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
- Lattice Insights for 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, October 2023**

| Section | Change Summary |
|---|---|
| All | Production release. |

www.latticesemi.com