



Composable Custom Extensions on Lattice RISC-V RX User Guide

Application Note

FPGA-AN-02075-1.2

September 2026

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

Abbreviations in This Document	6
1. Introduction	7
1.1. Naming Conventions	7
1.1.1. Nomenclature	7
1.1.2. Signal Names	7
2. Overview	8
3. CXU-LI: Interface between RISC-V and CXU	9
3.1. Signal Definitions	9
3.2. CXU Feature Levels	9
3.3. CXU-L2 in the Lattice Propel Builder Software	10
4. CXU Software Model	12
4.1. CSRs	12
4.1.1. mcx_selector CSR	12
4.1.2. cx_status CSR (Deprecated)	13
4.2. Instructions	13
4.2.1. R-type Encoding	14
4.2.2. I-type Encoding	14
4.2.3. Flex-type Encoding	15
4.2.4. Software Model and the CXU-LI	16
4.3. Firmware	17
4.3.1. Assembly Language	17
4.3.2. High-Level Language	18
4.3.3. Preprocessor Macros	22
5. Design Examples	28
5.1. SHA-3 and DSP CXU SoC Reference Design	28
5.2. Propel Builder Configuration	28
5.3. Generated Top-Level RTL	30
5.4. CXU Module Interface	31
5.5. Selecting the Active CXU	32
5.6. SHA-3 CXU Firmware Walkthrough	32
5.6.1. Keccak State Load and Store	32
5.6.2. Keccak-f[1600] Rounds	33
5.6.3. Software and CXU Benchmark	33
5.7. Legacy Example: Simple Endianness Conversion	34
5.7.1. Propel Builder Software	34
5.7.2. Top-Level RTL	35
5.7.3. Endianness Swapping CFU Module RTL	36
5.7.4. Example Firmware	38
References	39
Technical Support Assistance	40
Revision History	41

Figures

Figure 2.1. Multi-CXU System	8
Figure 3.1. RISC-V RX Configuration GUI.....	10
Figure 3.2. CXU Modules Connected in the Propel Builder RISC-V RX SHA-3 CXU Scalable Template	11
Figure 4.1 mcx_selector CSR 0xBC0 Version 0: Legacy Custom Instructions.....	12
Figure 4.2 mcx_selector CSR 0xBC0 Version 1: Extension Multiplexing	12
Figure 4.3. cx_status CSR (0x801) – Deprecated.....	13
Figure 4.4. CXU R-type Instruction Encoding	14
Figure 4.5. CXU I-type Instruction Encoding	15
Figure 4.6. CXU Flex-type Instruction Encoding.....	15
Figure 4.7. CXU Flex-type Instruction Alternate Encoding.....	15
Figure 4.8. Relationship between Instructions and CSRs of the CPU and the CXU-LI.....	16
Figure 4.9. Enabling the Preprocess Only Option	27
Figure 5.1. RISC-V RX CXU Configuration (Propel Builder Software 2026.1)	29
Figure 5.2. CXU Modules Connected to Ports M0 and M1 (Propel Builder Software 2026.1)	30
Figure 5.3. Generated Top-Level RTL (sha3_soc.v)	31
Figure 5.4. CXU Module Interface (sha3_cxu and dsp_acc_cxu)	31
Figure 5.5. Selecting the Active CXU (cxu.h and main.c)	32
Figure 5.6. Keccak State Load and Store (sha3_cxu.c)	32
Figure 5.7. Keccak-f[1600] Rounds (sha3_cxu.c)	33
Figure 5.8. Software and CXU-Accelerated SHA-3 Timing (main.c)	33
Figure 5.9. Enable CFU Port on RISC-V RX.....	34
Figure 5.10. RISC-V RX Design Exporting CFU-L2, Clock, and Reset	35
Figure 5.11 Top-Level RTL	36
Figure 5.12. Endianness Swapping CFU Module.....	37
Figure 5.13. Endianness Swapping Example Firmware	38

Tables

Table 3.1. CXU-LI Port Definitions.....9

Table 3.2. CXU Feature Levels.....9

Table 4.1. mcx_selector CSR Field Descriptions.....13

Table 4.2. cx_status CSR Field Descriptions.....13

Table 4.3. RISC-V ISA Opcodes (inst[1:0] = 11)14

Table 4.4. Opcode_I Macro Parameters25

Table 5.1. SHA-3 and DSP CXU SoC Modules28

Table 5.2. opcode_flex25 func3 Operations Used by the SHA-3 Firmware.....33

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
ANSI	American National Standards Institute
CF	Custom Function
CFU	Custom Function Unit (legacy term, see CXU)
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSR	Control and Status Register
CX	Composable Extension
CXE	Custom Operation Exception Enable
CXU	Composable Extension Unit
CXU-LI	Composable Extension Unit Logic Interface
DSP	Digital Signal Processor
GPIO	General Purpose Input/Output Processor
GUI	Graphical User Interface
HDL	Hardware Description Language
IP	Intellectual Property
ISA	Instruction Set Architecture
LED	Light-Emitting Diode
MSB	Most Significant Bit
RISC-V	An open-source instruction set architecture based on reduced instruction set computing (RISC) principles. This free and open standard enables a new era of processor innovation through open standard collaboration.
RTL	Register Transfer Level
SDK	Software Development Kit
SoC	System on Chip
UART	Universal Asynchronous Receiver/Transmitter

1. Introduction

Lattice Semiconductor RISC-V RX CPU includes an optional interface to a composable extension unit (CXU). This interface enables the addition of user-defined custom instructions that can be used to accelerate complex or regularly executed operations, thereby improving system performance.

The purpose of this application note is to provide an introduction on how to implement a CXU, connect it to the RISC-V RX core using the Lattice Propel™ Builder scalable template flow, and write firmware that executes CXU-specific instructions. A two-CXU SHA3 cryptographic hash plus DSP acceleration SoC is used as the main reference design throughout.

Note: The CXU-LI is supported in the Balanced and Advanced processor modes only. The Lite processor mode does not support CXU-LI.

1.1. Naming Conventions

1.1.1. Nomenclature

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

Note: This document uses the updated RISC-V Composable Custom Extensions Specification terminology. The former term "custom function unit" (CFU) has been replaced by "composable extension unit" (CXU), and "Custom Function Unit Logical Interface" (CFU-LI) has been replaced by "Composable Extension Unit Logic Interface" (CXU-LI). For more information, refer to the latest [RISC-V RX CPU IP User Guide \(FPGA-IPUG-02302\)](#) and the [RISC-V Composable Custom Extensions Specification](#).

1.1.2. Signal Names

- `_n` are active low signals (asserted when value is logic 0)
- `_i` are input signals
- `_o` are output signals

2. Overview

The Composable Custom Extensions Specification is an emerging industry standard. Lattice's RISC-V RX core implements a subset of the features described in the specification to enable the addition of custom accelerators while avoiding negative impacts on CPU core performance.

The Composable Custom Extensions Specification defines a CXU logical interface (CXU-LI) and protocol by which a RISC-V CPU can be attached to and communicate with one or more custom accelerator blocks.

Figure 2.1 shows the relationship between a Composable Custom Extensions-compliant RISC-V core, one or more CXUs, and the CXU-LI that connects them.

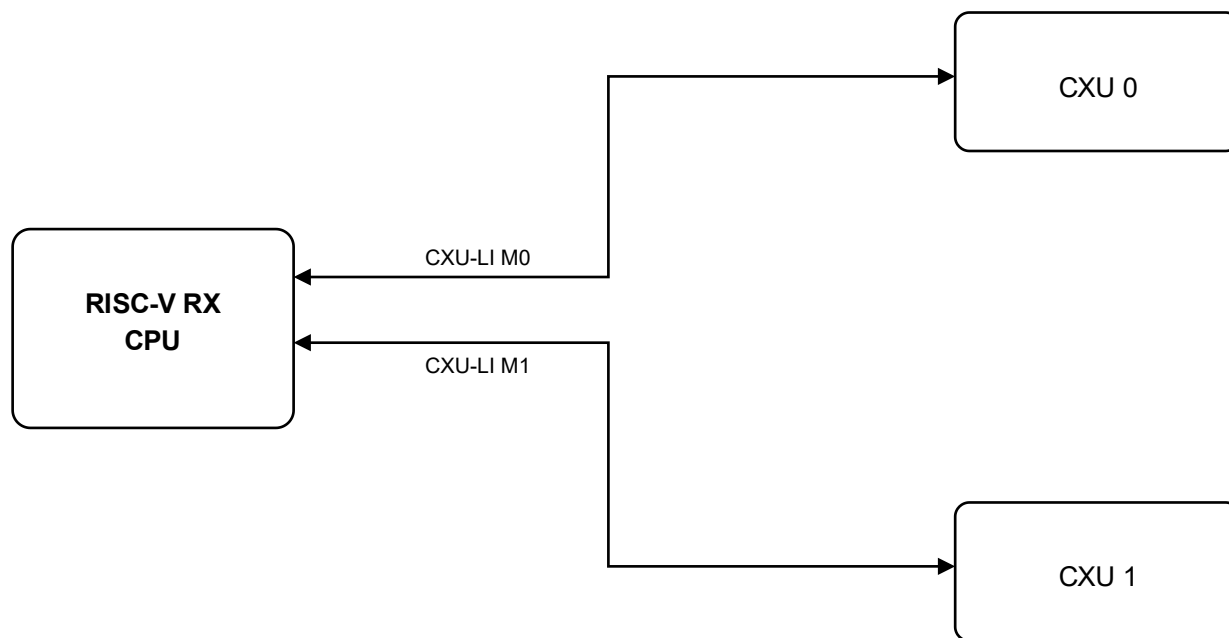


Figure 2.1. Multi-CXU System

The specification allocates three of the op codes that the RISC-V instruction set architecture (ISA) reserves for custom instructions. When executed by a Composable Custom Extensions-compliant core, these instructions cause the core to issue a request over the CXU-LI to the targeted CXU. A fourth custom opcode (custom-3) is reserved for CPU built-in custom instructions. For more information, refer to the *Set Reset Vector and Soft Reset Built-In Custom Instructions* section in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302).

A request generally includes two operands: either the contents of two of the registers in the CPU register file or a single register from the register file and an immediate value. The CXU-LI uses handshaking similar to AXI4 for both request and response transactions, and this allows a CXU to support fixed-length or variable-length processing delays. Once the CXU has processed a request, it responds with status information and an optional result. The result is stored in the RISC-V register file at a location specified by a field within the instruction that initiated the request.

The Composable Custom Extensions Specification also assigns a small subset of the Control and Status Registers (CSRs) that are reserved by the RISC-V ISA for the purpose of adding custom features. These CSRs are used to configure the CXU-LI, to select the current target CXU, and to select the CXU state context, if the implemented function is stateful and multiple threads are using it.

Note: The custom-3 opcode is reserved for CPU built-in custom instructions such as Set Reset Vector and Soft Reset (available in Advanced mode only). These built-in instructions are accessed when `mcx_selector.version=0`. For more information, refer to the *Set Reset Vector and Soft Reset Built-In Custom Instructions* section in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302).

3. CXU-LI: Interface between RISC-V and CXU

The CXU-LI is an interconnect and protocol by which the RISC-V issues requests to the CXU and the CXU communicates responses. In the CXU-LI system, the CPU is the requestor and the CXU is the responder. The CPU sends the CXU a request and eventually receives the CXU response. For each request, there is exactly one response.

3.1. Signal Definitions

The signals defined in the CXU specification are repeated in [Table 3.1](#).

Table 3.1. CXU-LI Port Definitions

Signal Name	Direction	Width	Level	Description
req_valid	CPU -> CXU	1	—	Request Valid: Handshaking signal indicating that the CPU is sending a request.
req_ready	CXU -> CPU	1	2+	Request Ready: Handshaking signal indicating that CXU is ready to receive a new request. Required for variable-latency CXU types.
req_cxu	CPU -> CXU	4	All	Request CXU ID: To identify which CXU is being requested in a multiple CXU design.
req_state	CPU -> CXU	3	1+	Request State ID: Selects which state to use when processing the request. For example, in cases where multiple threads use the same stateful CXU.
req_func	CPU -> CXU	4	All	Request Function ID: Selects which function to perform within the specified CXU (for example, load state vs. accumulate). The MSB is fixed to 0.
req_insn	CPU -> CXU	32	2+	Request Raw Instruction: The entire 32-bit instruction whose execution generated the request.
req_data0	CPU -> CXU	32	All	Request Operand Data 0: The first operand of the requested CXU operation.
req_data1	CPU -> CXU	32	All	Request Operand Data 1: The second operand of the requested CXU operation.
resp_valid	CXU -> CPU	1	1+	Response Valid: Handshaking signal from the CXU indicating that the CXU has completed the requested operation and that the result is available on resp_data.
resp_ready	CPU -> CXU	1	2+	Response Ready: Handshaking signal from the CPU indicating that it is ready to receive the CXU response.
resp_status	CXU -> CPU	3	All	Response Status: Carries a success or error code from the CXU in response to a request.
resp_data	CXU -> CPU	32	All	Response Data: The result of the CXU operation. Written to a register within the register file specified by the rd field of the instruction that initiated the request.

3.2. CXU Feature Levels

The Composable Custom Extensions Specification defines four feature levels (0–3), as shown in [Table 3.2](#). The RISC-V RX core supports interface level L2, with downward compatibility to support L0 and L1 as well. Consequently, the CXU ports on the RISC-V RX are of type CXU-L2.

Table 3.2. CXU Feature Levels

Feature Level	CXU Type	RISC-V RX Support
0	Combinational	Supported (degrade L2 by tying appropriate signals).
1	Fixed latency	Supported (degrade L2 by tying appropriate signals).
2	Variable latency	Natively supported.
3	Reordering	Not supported.

To degrade L2 to support a combinational (L0) or fixed-latency (L1) CXU, tie the following signals to constant values:

- For L0 (combinational): Assert `resp_valid` high whenever `req_valid` is high (combinational path). Tie `req_ready` high (always ready).
- For L1 (fixed latency): Tie `req_ready` high (always ready). Assert `resp_valid` after the fixed number of clock cycles following `req_valid`.

For detailed guidance, refer to the RISC-V Composable Custom Extensions Specification.

3.3. CXU-L2 in the Lattice Propel Builder Software

When a RISC-V RX is instantiated within the Lattice Propel Builder software, one to eight CXU-LI ports can be exported by enabling the checkbox for the **Enable CXU Ports** option and selecting the number of CXUs, as shown in Figure 3.1.

Note: The CXU-LI is only available when the processor is configured in the Balanced or Advanced mode.

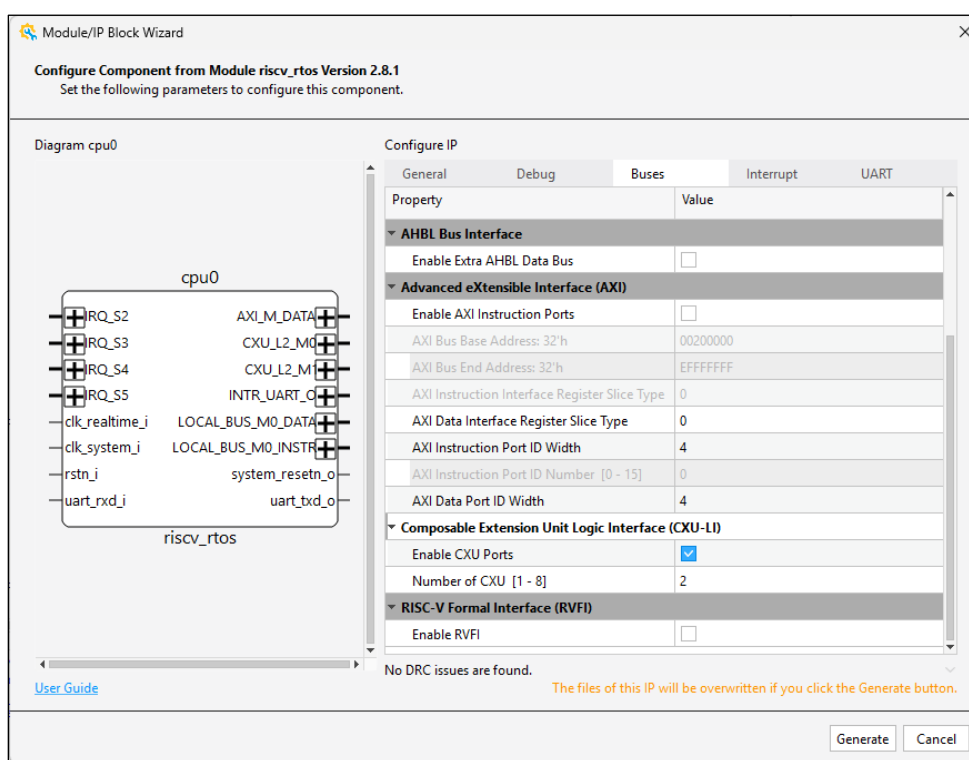


Figure 3.1. RISC-V RX Configuration GUI

In the Propel Builder scalable template flow, CXU modules are packaged as first-class Propel Builder IP modules and connected directly in the block diagram. The tool auto-generates all interconnect wiring between the CPU CXU-LI ports and each CXU module instance in the top-level SoC Verilog file. This is the recommended integration method; it avoids manual top-level RTL wiring and is shown in the [Design Examples](#) section.

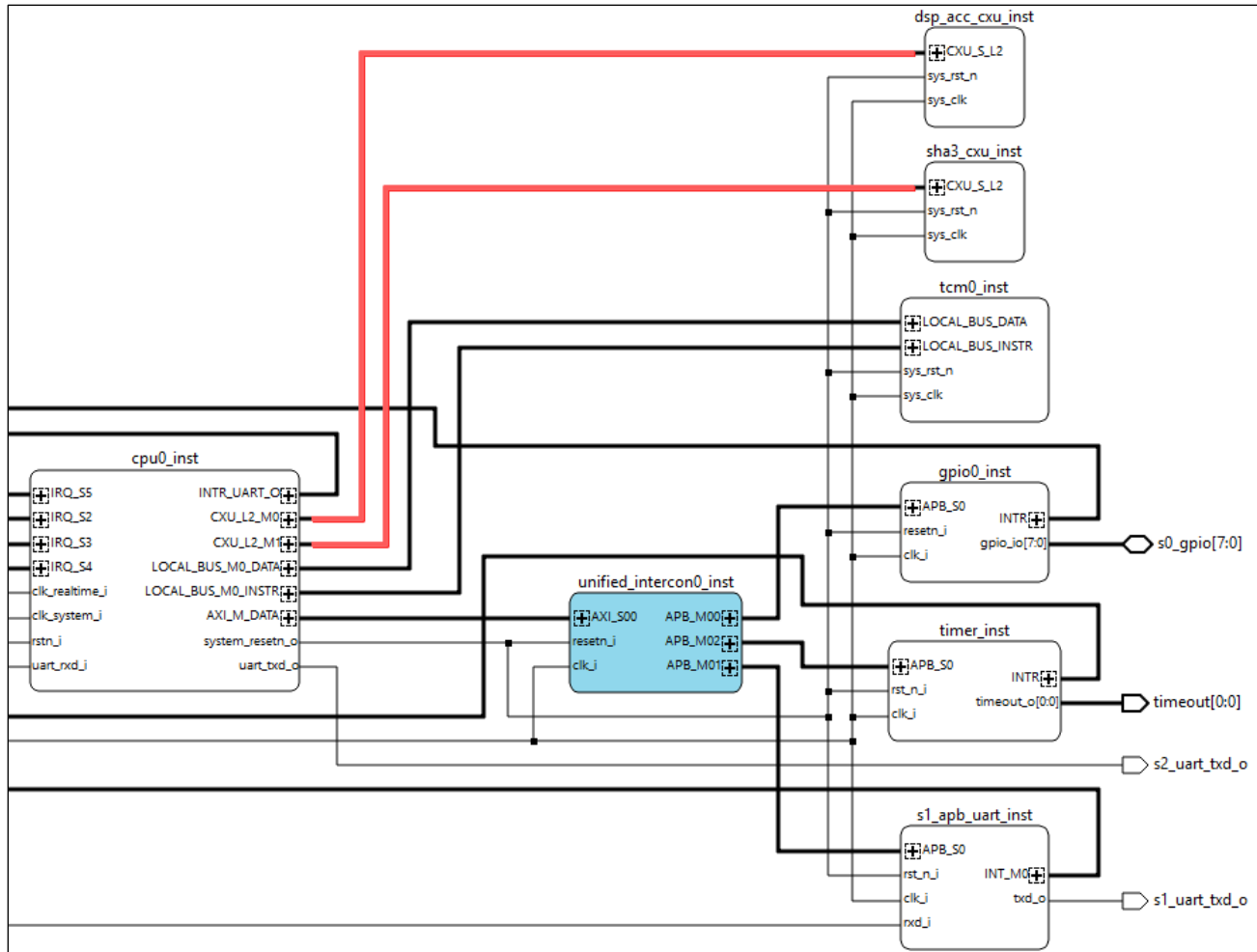


Figure 3.2. CXU Modules Connected in the Propel Builder RISC-V RX SHA-3 CXU Scalable Template

When version=1:

- cxe=0: custom-0/1/2 instructions issue CXU requests to the CXU and state context identified by cxu_id and state_id.
- cxe=1: custom-0/1/2 instruction accesses raise an illegal-instruction exception.

Table 4.1. mcx_selector CSR Field Descriptions

Field	Description
version	Extension multiplexing version (bits [31:29]). 0 = Legacy custom instructions (version 0). 1 = Extension multiplexing (version 1). 2-7 = Reserved.
cxe	Custom operation exception enable (bit[28]). 0 = Custom instructions are enabled. 1 = Custom instructions raise an illegal-instruction exception.
state_id	Selects the current hart CXU state context (bits [23:16], version 1 only). The lower three bits of this field will be driven on the <i>req_state</i> port of the selected CXU-L2 port during a CXU request.
cxu_id	Selects the current hart CXU (bits [7:0], version 1 only). The lower four bits of this field will be driven on the <i>req_cxu</i> port of the RISC-V's CXU-L2 port during a CXU request.

4.1.2. cx_status CSR (Deprecated)

The cx_status CSR (formerly cfu_status, CSR address 0x801) was defined in earlier versions of the Composable Custom Extensions Specification to accumulate error flags returned by the CXU(s) using six bits:

Table 4.2. cx_status CSR Field Descriptions

Field	Description
CU	Custom CXU operator error
OP	CXU operation error
FI	Invalid CXU function ID error
OF	Selected state context is in the off-state error
SI	Invalid CXU state ID error
CI	Invalid CXU ID error



Figure 4.3. cx_status CSR (0x801) – Deprecated

This CSR is not supported on the RISC-V RX CPU IP v2.8.0. The CXU-LI resp_status signal (3 bits) is used for per-request status reporting instead. Applications requiring status information from a CXU response should read the resp_status signal.

Refer to the *CXU-LI* section in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302) for the authoritative resp_status signal encoding on the RISC-V RX CPU IP v2.8.0.

4.2. Instructions

The Composable Custom Extensions Specification defines three general instruction encodings. The first two encodings are based on the RISC-V ISA's R-type and I-type instructions. The third encoding—flex-type—can also be mapped to the RISC-V's R-type instruction encoding although it does not expect a response.

The RISC-V ISA reserves four opcodes for custom instructions, as shown in [Table 4.3](#).

Table 4.3. RISC-V ISA Opcodes (inst[1:0] = 11)

inst [4:2]	000	001	010	011	100	101	110	111 (> 32b)
inst [6:5]								
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥ 80b

The Composable Custom Extensions specification uses three of these opcodes for custom function (CF) instructions:

- CUSTOM0 (0x0B) for the R-type CXU instruction (pseudo assembly: `cx_reg`)
- CUSTOM1 (0x2B) for the I-type CXU instruction (pseudo assembly: `cx_imm`)
- CUSTOM2 (0x5B) for the flex CXU instruction (pseudo assembly: `cx_flex / cx_flex25`)

Note: The CUSTOM3 (0x7B) opcode is reserved for CPU built-in custom instructions (for example, Set Reset Vector and Soft Reset in Advanced mode) and is not available for user CXU operations. These built-in instructions are accessed when `mcx_selector.version=0` and `mcx_selector.cxe=0`. For more information, refer to the *Set Reset Vector and Soft Reset Built-In Custom Instructions* section in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302).

4.2.1. R-type Encoding

The Composable Custom Extensions Specification R-type instruction is based on the RISC-V ISA Integer Register-Register instruction format. It takes two operands in the form of two indices—*rs1* and *rs2*—into the CPU register file. A third index into the register file—*rd*—specifies where the result of the operation is to be stored.

Pseudo assembly: `cx_reg cf_id, rd, rs1, rs2`

An R-type CF instruction issues a CXU request for a zero-extended 10-bit CF_ID *cf_id* with two source register operands identified by *rs1* and *rs2*. The CXU response data is written to the destination register *rd*.

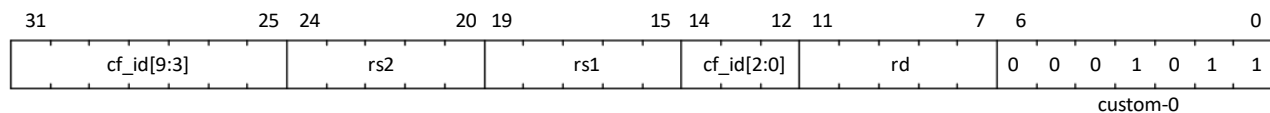


Figure 4.4. CXU R-type Instruction Encoding

4.2.2. I-type Encoding

The Composable Custom Extensions Specification I-type instruction is similar to the RISC-V ISA Integer Register-Immediate instruction format. It takes two operands in the form of an index—*rs1*—into the CPU's register file and a sign-extended 12-bit immediate value in the instruction *imm* field. The result is stored in the register file at the index specified by the *rd* field.

Pseudo assembly: `cx_imm cf_id, rd, rs1, imm`

An I-type CF instruction issues a CXU request for a zero-extended 3-bit CF_ID *cf_id* with one source register operand identified by *rs1* and a sign-extended 12-bit immediate value *imm*. The CXU response is written to the destination register *rd*.

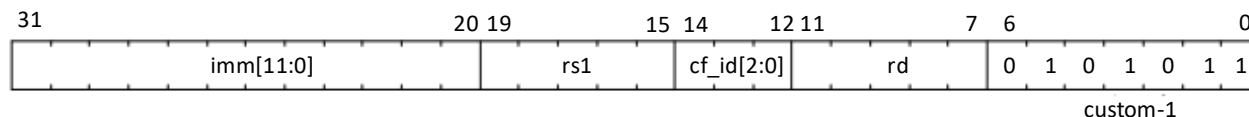


Figure 4.5. CXU I-type Instruction Encoding

Note: The I-type encoding has changed from the previous version of this document. Key differences:

- The immediate field is now 12 bits (bits [31:20]), up from 8 bits.
- The cf_id field is now 3 bits at position [14:12], down from 4 bits.
- The fixed “000” field previously at [14:12] has been replaced by cf_id[2:0].

4.2.3. Flex-type Encoding

The Composable Custom Extensions Specification flex-type encoding is used to initiate operations that do not require a response. Nothing is written back to the register file by a flex-type operation. There are two forms of the flex-type instruction encoding—a primary form in Figure 4.6 and an alternate form in Figure 4.7.

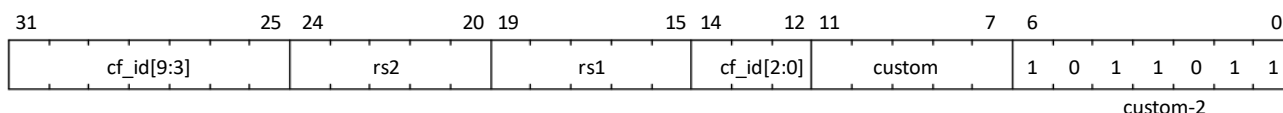


Figure 4.6. CXU Flex-type Instruction Encoding

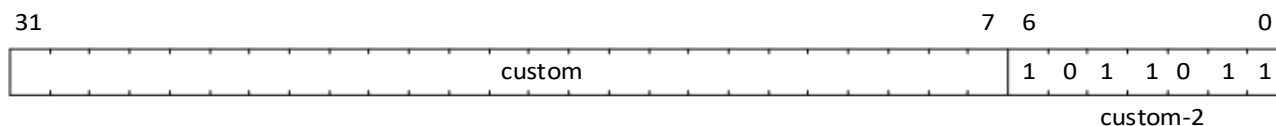


Figure 4.7. CXU Flex-type Instruction Alternate Encoding

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

4.2.4. Software Model and the CXU-LI

Figure 4.8 shows the relationship between the instructions and CSRs of the CPU and the CXU-LI.

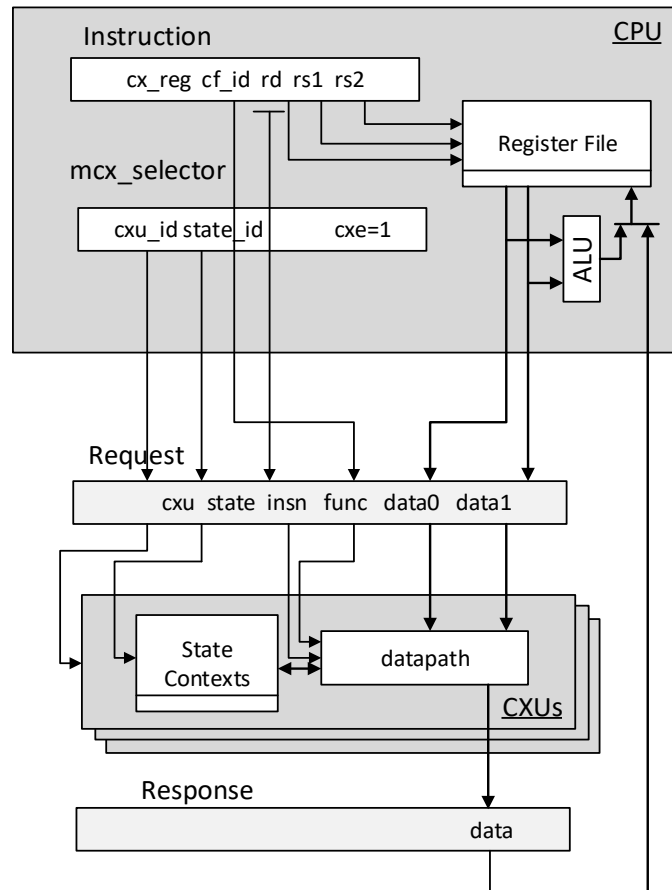


Figure 4.8. Relationship between Instructions and CSRs of the CPU and the CXU-LI

Note: Figure 4.8 shows the abbreviated signal labels—cxu, state, insn, func, data0, data1, data. The corresponding CXU-LI port names in the *CXU-LI* section of the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302) are: req_cxu, req_state, req_insn, req_func, req_data0, req_data1, resp_data, resp_status.

The *req_state* and *req_cxu* busses originate in the *mcx_selector* CSR. The *req_insn* bus is the entire 32-bit instruction word. The source of the *req_func* bus is the *cf_id* field of the instruction. The 32-bit *req_data0* and *req_data1* busses are sourced by the register file registers indexed by the *rs1* and *rs2* fields of the R-type instruction, respectively (for the I-type instruction, *req_data1* is sourced by the sign-extended *imm* field of the instruction).

The 32-bit result of the CXU operation is returned on the *resp_data* bus and is stored in the register file at the location indexed by the *rd* field of the instruction. For custom-2 flex-type instructions that do not write response data to the CPU register, the CXU only processes the request, possibly updating the selected state context. The response data is invalid for the CPU.

4.3. Firmware

4.3.1. Assembly Language

The Composable Custom Extensions Specification describes a new assembler pseudo instructions for issuing CXU custom function instructions. The four pseudo assembly mnemonics are:

- `cx_reg cf_id, rd, rs1, rs2` (R-type: custom-0 opcode)
- `cx_imm cf_id, rd, rs1, imm` (I-type: custom-1 opcode)
- `cx_flex cf_id, rs1, rs2` (Flex-type: custom-2 opcode)
- `cx_flex25 custom` (Flex-type alternate: custom-2 opcode)

Note that the standard GNU assembler that currently ships with the Propel SDK does not support these pseudo instructions natively.

CXU custom instructions can still be represented in assembly code. The GNU assembler implements two directives—`.insn` and `.word`—that can be used to insert arbitrary instruction into the assembly source code.

4.3.1.1. `.insn` Assembly Directive

The `.insn` directive has three forms:

- `.insn type, operand [, ..., operand_n]`
- `.insn insn_length, value`
- `.insn value`

The R and I versions of the first form are:

```
R-type:      .insn r opcode, func3, func7, rd, rs1, rs2
I-type:      .insn i opcode, func3, rd, rs1, imm12
```

These can be useful for issuing the R-type and I-type CFU instructions. For example, an R-type CXU instruction can be written in assembly as:

```
.insn r 0x0B, cf_id[2:0], cf_id[9:3], rd, rs1, rs2
```

Where:

`r` – denotes the type of encoding (R-type)
`0x0B` – is the opcode for the R-type CXU instruction (custom-0)
`cf_id[2:0]` – lower 3 bits of the 10-bit Custom Function ID, placed in the func3 field at bits [14:12]
`cf_id[9:3]` – upper 7 bits of the 10-bit Custom Function ID, placed in the func7 field at bits [31:25]
`rd` – specifies the destination register
`rs1` – specifies the first source register
`rs2` – specifies the second source register

Similarly, an I-type instruction could be written as:

```
.insn i 0x2B, cf_id, rd, rs1, imm
```

Where:

`i` – denotes the type of encoding (I-type)
`0x2B` – is the opcode for the I-type CXU instruction. (custom-1)
`cf_id` – is the 3-bit Custom Function ID [2:0], placed in the func3 field at bits [14:12]
`rd` – specifies destination register.

rs1 – specifies the source register.

imm – is the 12-bit sign-extended immediate value placed at bits [31:20].

The primary form of the CXU flex-type instruction uses the same bit layout as the R-type instruction, except that the CXU response data is not written back to the CPU's register file. In place of the destination register field (rd), the flex-type primary form carries a 5-bit user-defined custom field. It can be encoded in assembly using the GNU ``as` .insn r`` directive as:

```
.insn r 0x5B, cf_id[2:0], cf_id[9:3], custom, rs1, rs2
```

Where:

r – denotes the `.insn` directive form (R-type bit layout, used to emit the flex-type primary encoding)

0x5B – is the opcode for the flex-type CXU instruction (custom-2).

cf_id[2:0] – lower 3 bits of the 10-bit Custom Function ID, placed in the func3 field at bits [14:12]

cf_id[9:3] – upper 7 bits of the 10-bit Custom Function ID, placed in the func7 field at bits [31:25].

custom – 5-bit user defined field placed at bits [11:7] (the rd slot is repurposed; no register write-back)

rs1 – specifies the first source register.

rs2 – specifies the second source register.

4.3.1.2. .word Assembly Directive

The nominal purpose of the `.word` assembly directive is to insert 32-bit data words into assembly source. However, it can also be used to insert 32-bit instructions.

The `.word` directive has the following form:

```
.word value
```

Where:

value – is a 32-bit value to be inserted into the assembly code.

Note that the assembler can evaluate C-like expressions so value can be specified in terms of shifted and bit-wise OR'd constants. For example, the alternate version of the CFU flex-instruction can be written in assembly using the `.word` pseudo directive:

```
.word (custom << 7 | 0x5B)
```

Where:

custom – is the 25-bit immediate custom field of the CFU flex-type instruction.

0x5B – is the opcode of the CFU flex-type instruction.

The `.set` assembly directive can be used to set the value of a symbol. Note that `.set` does not emit any code. In the example above, the symbol, 'custom' could be set to a value as follows:

```
.set custom, 0x123456
```

4.3.2. High-Level Language

The standard GNU C/C++ compiler that ships with the Propel SDK cannot translate high-level source into instructions that take advantage of custom accelerator logic. Modifying the compiler to do so is beyond the scope of this document.

4.3.2.1. Inline Assembly

If high-level source code such as C or C++ is to issue custom CFU instructions, it must do so using inline assembly. The GNU C extended `asm` keyword can be used to facilitate insertion of assembly statements into C/C++ source code.

There are two forms of an extended `asm` statement—a form that supports “goto” operations (for example, jumps or calls) and a form that does not support “goto” operations. For the purposes of this application note, only the latter form is relevant.

`asm` syntax with output operands and not gotos:

```
asm asm-qualifiers (AssemblerTemplate
                    :OutputOperands
                    [:InputOperands
                    [:Clobbers]])
```

Where:

<i>asm-qualifiers</i> –	One or more of the keywords: <code>volatile</code> , <code>inline</code> , <code>goto</code> . It is recommended that the <code>volatile</code> qualifier be used in CFU applications to prevent the compiler from optimizing out the inline assembly instruction(s). The <code>goto</code> qualifier is not expected to be used in CFU-related operations.
<i>AssemblerTemplate</i> –	A string literal describing the assembly code instruction. Similar to a <code>printf</code> format string, the <i>AssemblerTemplate</i> contains a mix of fixed text and tokens that refer to input and output parameters.
<i>OutputOperands</i> –	A comma-separated list of C/C++ variables that are modified by the instruction(s) in the <i>AssemblerTemplate</i> . For a single CFU instruction, this would be a single variable (R-type and I-type instructions) or no variables (flex-type instruction).
<i>InputOperands</i> –	An optional comma-separated list of C/C++ variables that are read by the instruction(s) in the <i>AssemblerTemplate</i> . For a single CFU instruction, this would be one variable (I-type instruction), two variables (R-type and flex-type instructions) or no variables (alternate flex-type encoding).
<i>Clobbers</i> –	An optional comma-separated list of registers that are changed (“clobbered”) by the <i>AssemblerTemplate</i> , beyond those listed as outputs. For CFU instructions, this is expected to be an empty list.

Note: For ANSI C, the `asm` keyword should be both preceded and followed by double underscores. For example, “`__asm__`”.

A deep discussion of all the options available for the *AssemblerTemplate* is beyond the scope of this document. For the purposes of writing RISC-V CFU inline assembly, note that tokens in the *AssemblerTemplate* string are prefixed by a percent sign, ‘%’. Tokens can refer to elements in the input and output operand lists either by numeric position in the arguments lists (For example, %0, %1, and so on) or by a symbolic name. This document uses symbolic names.

Symbolic names are enclosed by square brackets (For example, [my_operand]) in both the *AssemblerTemplate* string and in the respective *OutputOperands* or *InputOperands* list. Each element in those lists also contains a constraint enclosed in double quotation marks and the name of the associated C variable enclosed in parenthesis.

For CFU operations, an output constraint is typically going to be “=r”. The equals sign, ‘=’, means that the value in the variable is overwritten and the ‘r’ means that the result is to be placed in a register. For input operations, the constraint is typically going to be “r”, which means that the source of the data is a register.

So, a single element in an *OutputOperands* list might look like:

```
[res]”=r”(result)
```

And an element in the *InputOperands* list might look like:

```
[rs1]”r”(input_argument)
```

4.3.2.2. I-Type Instruction Inline Assembly Example

Putting this all together to execute an I-type instruction from C source such that:

Function ID:	3
Immediate Value:	12
Source:	C variable, unsigned int arg
Destination:	C variable, unsigned int result

Write the following C code:

```
// Execute an I-type CXU instruction:
//   Function ID: 3
//   Immediate Value: 12
//   Source: C variable, unsigned int arg
//   Destination: C variable, unsigned int result

unsigned int arg=10;
unsigned int result;
__asm__ volatile (".insn i 0x2B, 3, %[rd], %[rs1], 12"
    : [rd]"=r"(result)
    : [rs1]"r"(arg)
);

// In the new encoding:
//   0x2B      = custom-1 opcode (bits [6:0])
//   3         = cf_id[2:0] placed in func3 field (bits [14:12])
//   12        = 12-bit immediate value (bits [31:20])
```

4.3.2.3. R-Type Instruction Inline Assembly Example

Similarly, if an R-type instruction was to be executed from C source:

Function ID:	1
Source 1	C variable, unsigned int arg1
Source 2:	C variable, unsigned int arg2
Destination:	C variable, unsigned int result

Write the following code:

```
unsigned int arg1=0x44001100;    // some arbitrary input
unsigned int arg2=25;           // some arbitrary input
unsigned int result;            // variable to store the result of the cfu computation

__asm__ volatile (".insn r 0x0B, 1, 0, %[rd], %[rs1], %[rs2]"
```

```

: [rd]"=r"(result)
: [rs1]"r"(arg1), [rs2]"r"(arg2)
);

```

4.3.2.4. Flex-Type Instruction Inline Assembly Example

The flex-type encoding will be similar to the R-type inline assembly command. The difference is that the *AssemblerTemplate* string will explicitly reference a register and the *OutputOperands* list will be empty. The number of the destination register will be the value that appears in the flex-type encoding's 5-bit custom field. For a flex-type instruction:

```

Function ID:      1
Source 1         C variable, unsigned int arg1
Source 2:        C variable, unsigned int arg2
Custom Field Value: 17

```

Write the following code:

```

unsigned int arg1=0x44001100; // some arbitrary input
unsigned int arg2=25;         // some arbitrary input

__asm__ volatile (".insn r 0x5B, 1, 0, x17, %[rs1], %[rs2]"
:
: [rs1]"r"(arg1), [rs2]"r"(arg2)
);

```

4.3.2.5. CXU CSR Access Inline Assembly

High-level source must also use inline assembly to read, write, or modify CSRs, including the CXU-related CSRs. For example, to write to the `mcx_selector` CSR, use the RISC-V assembly pseudo instruction, `csrw` (CSR write):

```

// mcx_selector CSR layout (version 1):
// bits [31:29] = version (001 for extension multiplexing)
// bit [28]     = cxe (0 = custom instructions enabled)
// bits [23:16] = state_id
// bits [7:0]   = cxu_id

unsigned int version = 1; // version 1 = extension multiplexing
unsigned int cxe     = 0; // 0 = custom instructions enabled
unsigned int cxu_id  = 0;
unsigned int state_id = 5;
unsigned int config = (version << 29) | (cxe << 28) | (state_id << 16) | (cxu_id << 0);

__asm__ volatile ("csrw 0xBC0, %[rs1]" // mcx_selector CSR address=0xBC0

```

```

:                                     // empty OutputOperands list
: [rs1]"r"(config)                   // use C variable, config, as input
);

```

A read from the `cx_status` CSR can be performed using the, `csrr` (CSR read) pseudo-instruction:

```

unsigned int status;

__asm__ volatile ("csrr %[res], 0x801" // cx_status CSR address=0x801
                  : [res]="r"(status)  // read result stored in "status"
                  );                  // no input arguments

```

Note: The `cx_status` CSR (0x801) is not supported on the RISC-V RX CPU IP v2.8.0. Use the 3-bit `resp_status` signal on the CXU-LI for per-request status reporting. For more information, refer to the *CXU-LI* section of the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302).

4.3.3. Preprocessor Macros

The C/C++ preprocessor operates through string substitution and can be useful when there is a need to programmatically build string constants at compile time.

Inline assembly only facilitates the insertion of the `rs1`, `rs2`, and `rd` fields into the Assembly Template string. Preprocessor macros enable the insertion of other required substrings such as the value of the opcode and the Custom Function ID (`cf_id`).

The SHA-3 Template that ships with the Propel Builder 2024.1 and later makes use of macros in its example firmware. This subsection reviews relevant aspects of preprocessor macros for those who may be unfamiliar with the C/C++ language.

4.3.3.1. Macro Definitions

C/C++ preprocessor macros are created using the `#define` preprocessor directive. The `#define` directive has two forms:

```

#define identifier [token-string]
#define identifier(param1, param2, ...) [token-string]

```

The second form creates a function-like macro that is useful for simplifying the construction of inline assembly calls:

<i>identifier</i> –	The macro name. Every occurrence of the identifier string that follows the macro definition will be “expanded” by replacing the identifier with the token-string.
<i>param1, param2,...</i> -	Optional parameters. An argument placed in the position of a given parameter will be used to replace any instance of the parameter name within the token-string.
<i>token-string</i> -	A fragment of valid C code which replaces each instance of the identifier that comes after the macro’s definition. Any occurrences of a parameter name within the body of the token-string will be replaced by the argument in that parameter position within the parameter list.

4.3.3.2. Using Macros to Create Strings

For example, a macro that builds a greeting string can be written as:

```
#define GREETING(name) "Hello " name "!"
```

and can be used in subsequent code:

```
GREETING("John")
```

Which expands to:

```
"Hello " "John" "!"
```

The parameter, *name*, is replaced by the argument, "John", at *name*'s position in the parameter list.

The compiler will interpret the successive double quoted strings as a single string:

```
"Hello John!"
```

If the parameter in the macro definition is preceded by a hash character, #, then the argument will automatically be enclosed in double quotes during macro expansion. This allows invocations of the macro to be simplified by omitting the double quotes around the argument. So, the new macro definition would be:

```
#define GREETING(name) "Hello " #name "!"
```

and the new invocation would look like:

```
GREETING(John)
```

4.3.3.3. Arithmetic Expressions within Macros

Macros can also be defined as arithmetic expressions. For example, a macro that finds the minimum value of two expressions can be written as:

```
#define MIN(A, B) ( (A) < (B) ? (A) : (B) )
```

and can be used in subsequent code:

```
z = MIN(x, y);
```

This statement is expanded by the preprocessor and passed to the compiler as:

```
z = ( (x) < (y) ? (x) : (y) );
```

The use of parentheses around the parameters in the token-string as well as the token-string itself ensures that order of operations is preserved. For example, in C/C++ the less than comparison operator has higher precedence than bitwise operations such as bitwise AND.

```
z = MIN( x & 0x05, y );
```

If the parentheses around the parameters were omitted from the macro definition above, the macro would expand to the following:

```
z = x & 0x05 < y ? x & 0x05 : y;
```

The compiler performs the comparison, `0x05 < y`, and then bitwise AND the result with the variable `x`, which is not what you would intuitively expect from the macro. The use of parentheses in the macro definition avoids this potential problem.

4.3.3.4. Code Blocks within Macros

A macro can consist of multiple C/C++ statements within a code block. For example, the MIN macro can be rewritten using an if-else statement instead of the ternary operator:

```
#define MIN2(A,B) \
({ \
    int result; \
    if ( (A) < (B) ) \
        result = (A); \
    else \
        result = (B); \
    result; \
})
```

where the backslash character, `\`, serves as the line continuation character.

The GNU C/C++ compiler supports a non-standard feature called “statement expressions” that allows a code block to take on the value of its last expression. This allows a code block to be used on the right-hand side of an assignment statement. In the MIN2 macro above, the last expression is the result variable. So, the MIN2 macro can be used to determine the smaller of two integer expressions, like the MIN macro, shown earlier.

```
z = MIN2(x, y);
```

4.3.3.5. SHA-3 Template Macros

The Composable Custom Extensions Specification defines its I-type instruction as shown in [Figure 4.5](#). The I-type instruction contains five fields: `imm[11:0]`, `rs1`, `cf_id[2:0]` (placed in the `func3` slot at bits [14:12]), `rd`, and `opcode`. The `opcode_I` macro parameter list corresponds directly with four of these five fields:

- `opcode`: The 7-bit RISC-V custom opcode (custom-1 = 0x2B for I-type CXU instructions). Passed as the first macro parameter.
- `func3`: The 3-bit Custom Function ID (`cf_id[2:0]`). The caller passes this at the `func3` parameter position; per the new I-type encoding, the `func3` slot at bits [14:12] is `cf_id[2:0]`.
- `rs1`: The first source register (5 bits at [19:15]). Supplied through the C-level variable bound to `[arg1]`.
- `imm`: The sign-extended 12-bit immediate (bits [31:20]). Passed as the fourth macro parameter.

The destination register `rd` is not a macro parameter. It is supplied by the C compiler through the `[result]` operand binding in the inline-assembly template (`regnum_%[result] << 7`), so the macro returns the result through the C expression rather than taking `rd` as an explicit argument. The SHA-3 template defines the `opcode_I` macro as follows:

```
#define opcode_I(opcode, func3, rs1, imm) \
({ \
    register unsigned long result; \
    asm volatile( \
        ".word ((\" #opcode \" ) | \
```

```

    (regnum_%[result] << 7) |          \
    (regnum_%[arg1] << 15) |          \
    ((" #imm ") << 20) |              \
    ((" #func3 ") << 12));\n"         \
    : [result] "=r" (result)          \
    : [arg1] "r" (rs1)                \
    );                                \
    result;                           \
})

```

Table 4.4. Opcode_I Macro Parameters

Bits	Field	Shift	Macro Source
[6:0]	opcode	0	`opcode` parameter
[11:7]	rd	7	`regnum_%[result]`
[14:12]	func3	12	`func3` parameter (caller passes cf_id[2:0])
[19:15]	rs1	15	`regnum_%[arg1]`
[31:20]	imm[11:0]	20	`imm` parameter

```
opcode_I(opcode, func3, rs1, imm)
```

The destination register field, *rd*, is the holding register that the compiler associates with the variable named *result*.

```
register unsigned long result;
```

The macro uses inline assembly to insert a single, custom instruction via the *asm* keyword.

```
asm volatile(
```

Within the assembly template, the *.word* directive is used to build the custom instruction.

```
".word (
```

Note that every occurrence of a parameter (for example, *opcode*, *imm*, and *func3*) is preceded by the hash or pound symbol, *#*. Wherever the macro is invoked, those parameters will be replaced by the invocation's corresponding arguments and they'll be enclosed in double quotes. After preprocessing, the inline assembly code inside of the assembly template will consist of a series of double quoted substrings. The compiler concatenates these substrings into a single string.

The *.word* assembly directive expects a purely numeric argument. However, the inline assembly function inserts register names (for example, *x1*, *a2*, *t4*, and so forth) into the resulting assembly code and these register names are alphanumeric.

The SHA-3 template reconciles this problem by defining a set of symbols that it uses to translate the register names into their corresponding numeric register file indices. The symbol names are the names of the registers, prefixed by the "regnum_" string. The symbol values are the indices. For example, the symbol for register *x2* is "regnum_x2". The "regnum_x2" symbol is assigned a value of 2, the position of *x2* within the register file.

Because this translation occurs during the assembly phase, the SHA-3 template defines the symbols using the ".set" assembler directive from within the inline assembly statements:

```
asm(".set regnum_x0 , 0");
asm(".set regnum_x1 , 1");
asm(".set regnum_x2 , 2");
```

```
asm(".set regnum_x3 , 3");
asm(".set regnum_x4 , 4");
asm(".set regnum_x5 , 5");
...
```

For example, if the compiler chose register x5 to hold the value of the variable, `result`, the portion of the macro's token string,

```
(regnum_%[result] << 7)
```

would be output by the preprocessor as:

```
(regnum_x5 << 7)
```

And, because the `regnum_x5` symbol is defined with a value of 5, the assembler interprets the subexpression as equivalent to:

```
(5 << 7)
```

Where 5 is the register index of x5 and it is left shifted by 7 bits to the location of *rd*, the I-type instruction's destination register field.

In addition to the translation symbols, the SHA-3 template also defines symbols corresponding to the four custom instruction opcodes: CUSTOM0 for R-type instructions, CUSTOM1 for I-type instructions, CUSTOM2 for flex instructions, and CUSTOM3 (reserved for CPU built-in custom instructions; not used by user-defined CXU instructions).

```
asm(".set CUSTOM0 , 0x0B");
asm(".set CUSTOM1 , 0x2B");
asm(".set CUSTOM2 , 0x5B");
asm(".set CUSTOM3 , 0x7B");
```

Putting this all together, the `opcode_I` macro can be used to simplify the insertion of CXU instructions. For example,

```
result = opcode_I(CUSTOM1, 1, input_arg, 213);
```

Where:

- CUSTOM1:** The RISC-V ISA custom-1 opcode (0x2B) for I-type instructions. Defined in ``cxu.h`` through ``asm(".set CUSTOM1, 0x2B");``
- 1:** The value placed in the func3 field at bits [14:12]. Per the new I-type encoding, this is the 3-bit Custom Function ID (`cf_id[2:0]`); selects the desired custom function within the CXU.
- input_arg:** A variable whose value will be placed in the register pointed at by the rs1 field. That value will also be driven on the req_data0 bus when the instruction request is sent to the CXU.
- 213:** The value of the imm parameter is placed at bits [31:20] of the instruction. The 12-bit imm field will be sign-extended to 32 bits and driven on the req_data1 bus during the transaction request to the CXU.

4.3.3.6. Debugging Macros

C/C++ preprocessor macros can be difficult to debug. One helpful debug strategy for macros is to examine the preprocessed source code. This can be achieved within the Propel SDK by checking the **Preprocess only (-E)** option as shown in Figure 4.9.

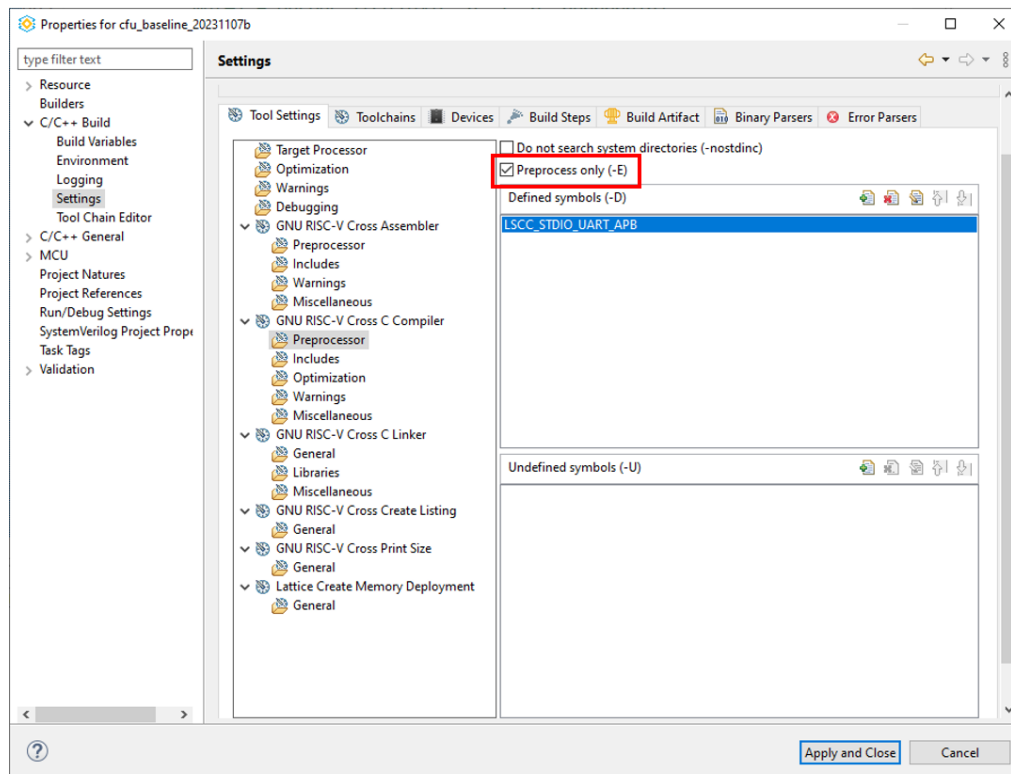


Figure 4.9. Enabling the Preprocess Only Option

When the **Preprocess only** option is enabled, the build process will halt following the preprocessor stage and the resulting C/C++ source code will be written to files with the *.o file extension. The preprocessed source can be examined to verify that the macro produces the desired C/C++ statement or expression.

5. Design Examples

5.1. SHA-3 and DSP CXU SoC Reference Design

This section presents a reference design that attaches two Composable Extension Units (CXUs) to a single RISC-V RX CPU: a SHA-3 cryptographic hash accelerator and a DSP accelerator used for CNN convolution. The design is built with the Lattice Propel Builder scalable template flow in Lattice Radiant™ 2026.1 and targets a Lattice CertusPro™-NX device. Each accelerator is packaged as an independent CXU module attached to its own CXU-LI port, so firmware can switch between them at run time by writing the `mcx_selector` CSR. Table 5.1 summarizes the modules in the reference SoC.

Table 5.1. SHA-3 and DSP CXU SoC Modules

Module	IP Version	CXU Port	Role
cpu0 (RISC-V RX CPU)	2.8.1	— (host)	CXU-LI requestor; issues CXU instructions and selects the active port through <code>mcx_selector</code> (0xBC0)
dsp_acc_cxu	1.0.0	M0 (cxu_id 0)	Signed 16x16 multiply-accumulate; accelerates CNN convolution
sha3_cxu	1.0.1	M1 (cxu_id 1)	Keccak-f[1600] state operations; accelerates SHA-3
tcm0	1.5.4	—	Tightly-coupled memory
unified_intercon0	2.0.0.08	—	System interconnect
gpio0	1.8.1	—	GPIO / LED output
pll0	1.9.1	—	System clock generation
timer	1.4.0	—	General-purpose timer (benchmark timebase)
s1_apb_uart	1.5.0	—	Console UART
osc_cre0	1.4.0	—	On-chip oscillator / configuration

Target device: LFCPNX-100-9LFG672C (CertusPro-NX). Tools: Lattice Radiant software 2026.1 and Propel Builder software 2026.1. The RISC-V RX CPU (cpu0) is the CXU-LI requestor; the two CXU modules are the responders. The DSP accelerator is attached to the CXU port M0 (cxu_id 0) and the SHA-3 accelerator to the CXU port M1 (cxu_id 1).

5.2. Propel Builder Configuration

The reference SoC is created with the Propel Builder software scalable template flow. In the RISC-V RX configuration GUI, the CXU interface is enabled in Advanced mode by selecting Enable CXU Ports and setting Number of CXU to 2. In the scalable template flow each CXU is a first-class Propel Builder software module: the `dsp_acc_cxu` module is connected to CXU port M0 and the `sha3_cxu` module to CXU port M1. Propel Builder automatically generates the CXU-LI interconnect between the CPU and each CXU module, as shown in Figure 5.1 and Figure 5.2.

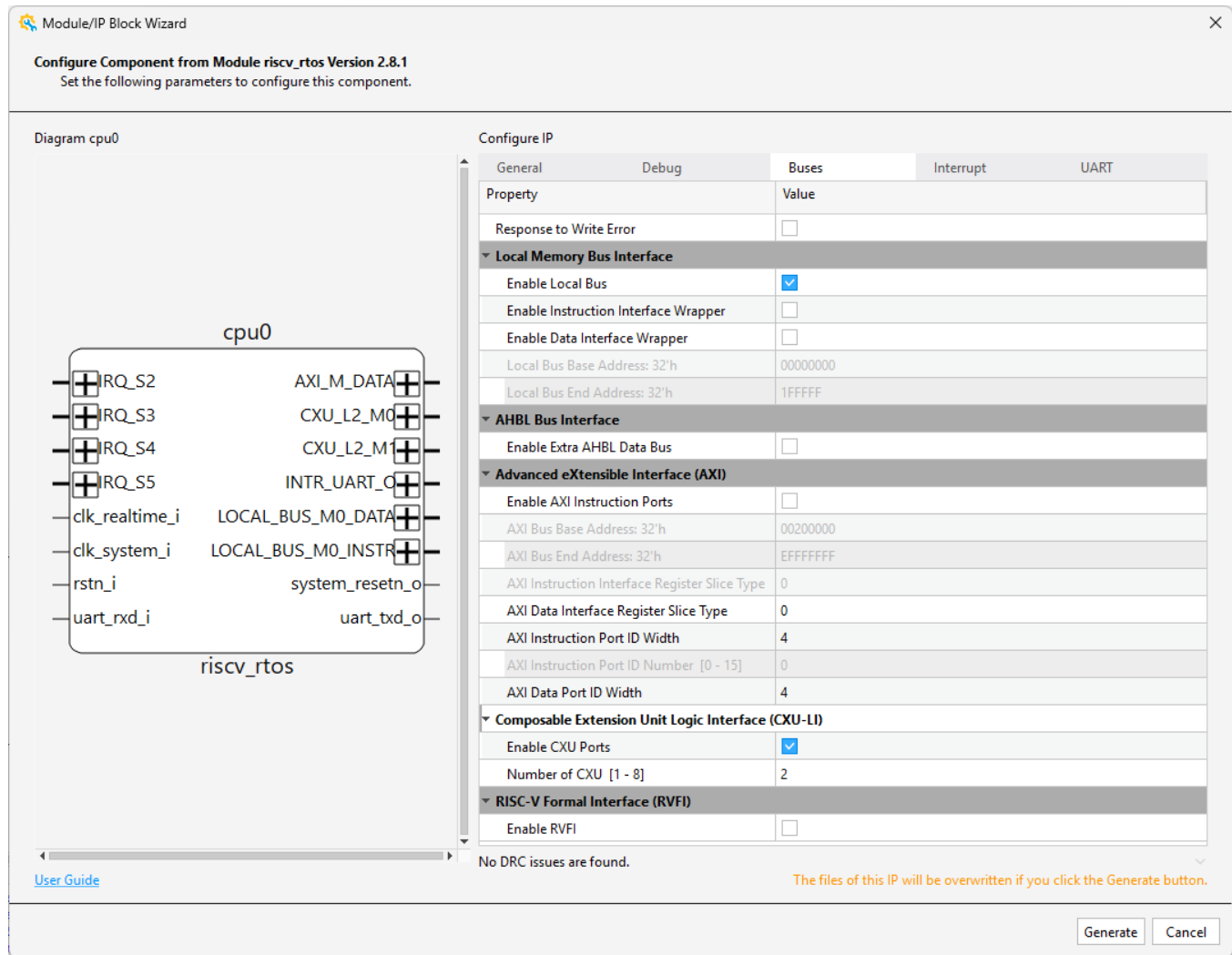


Figure 5.1. RISC-V RX CXU Configuration (Propel Builder Software 2026.1)

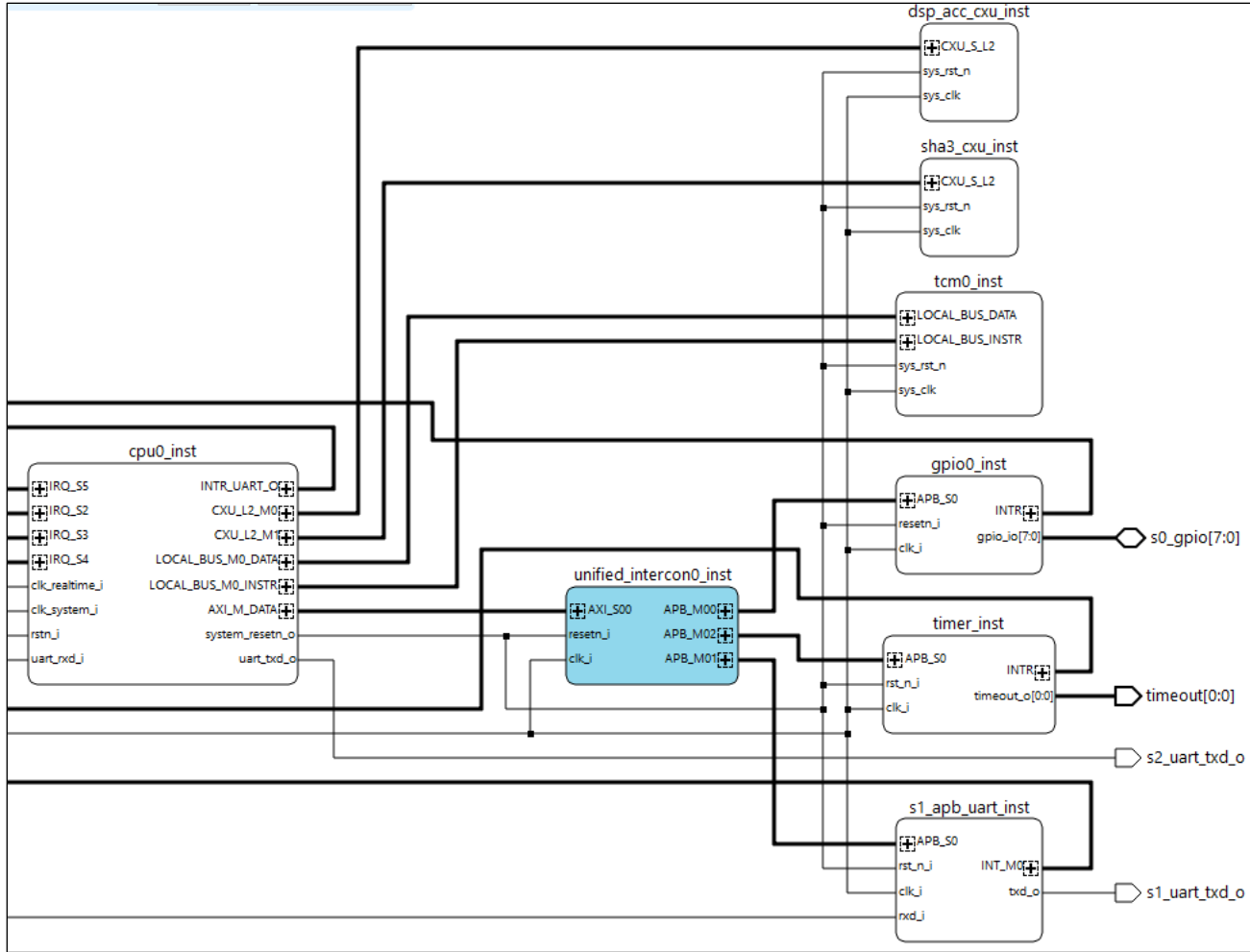


Figure 5.2. CXU Modules Connected to Ports M0 and M1 (Propel Builder Software 2026.1)

5.3. Generated Top-Level RTL

When the design is generated, the Propel Builder software produces the top-level Verilog module *sha3_soc.v*. It instantiates the CPU (cpu0), the two CXU modules, and the interconnect, and wires each CXU to its port. Figure 5.3 shows the relevant portions of the generated top level. Note that the generated CXU-LI signals use the port name *req_cfu*; the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302) refers to this signal as *req_cxu*. The two names denote the same signal (see Section 3.1).

```
module sha3_soc (s0_gpio, timeout, rstn_i, s1_uart_rxd_i, s1_uart_txd_o,
                s2_uart_rxd_i, s2_uart_txd_o);
    ...
    // CXU-LI interconnect wires: port M0 -> dsp_acc_cxu, port M1 -> sha3_cxu
    wire [3:0]cpu0_inst_CXU_L2_M0_interconnect_req_cfu;
    wire [3:0]cpu0_inst_CXU_L2_M1_interconnect_req_cfu;
    wire [3:0]cpu0_inst_CXU_L2_M0_interconnect_req_func;
    wire [2:0]cpu0_inst_CXU_L2_M0_interconnect_req_state;
    wire [31:0]cpu0_inst_CXU_L2_M0_interconnect_resp_data;
    wire [2:0]cpu0_inst_CXU_L2_M0_interconnect_resp_status;
    ... // req_data0/1, req_insn and the M1 equivalents are generated likewise

    // CPU exposes each CXU-LI port as _0 (M0) / _1 (M1) suffixed signals
```

```

cpu0 cpu0_inst (
    .req_cfu_0({cpu0_inst_CXU_L2_M0_interconnect_req_cfu}),
    .req_cfu_1({cpu0_inst_CXU_L2_M1_interconnect_req_cfu}),
    .req_func_0({cpu0_inst_CXU_L2_M0_interconnect_req_func}),
    .req_func_1({cpu0_inst_CXU_L2_M1_interconnect_req_func}),
    ...
    .resp_status_0({cpu0_inst_CXU_L2_M0_interconnect_resp_status}),
    .resp_status_1({cpu0_inst_CXU_L2_M1_interconnect_resp_status}), ... );

// DSP accelerator on port M0
dsp_acc_cxu dsp_acc_cxu_inst (
    .req_cfu({cpu0_inst_CXU_L2_M0_interconnect_req_cfu}),
    .req_func({cpu0_inst_CXU_L2_M0_interconnect_req_func}),
    .resp_data({cpu0_inst_CXU_L2_M0_interconnect_resp_data}),
    .resp_status({cpu0_inst_CXU_L2_M0_interconnect_resp_status}), ...
    .sys_clk(pll0_inst_clkop_o_net), .sys_rst_n(cpu0_inst_system_resetsn_o_net));

// SHA-3 accelerator on port M1
sha3_cxu sha3_cxu_inst (
    .req_cfu({cpu0_inst_CXU_L2_M1_interconnect_req_cfu}),
    .req_func({cpu0_inst_CXU_L2_M1_interconnect_req_func}), ...
    .sys_clk(pll0_inst_clkop_o_net), .sys_rst_n(cpu0_inst_system_resetsn_o_net));
endmodule

```

Figure 5.3. Generated Top-Level RTL (sha3_soc.v)

5.4. CXU Module Interface

Each CXU module exposes a CXU-LI (Level 2) port. The sha3_cxu and dsp_acc_cxu modules share an identical interface, shown in Figure 5.4. The request channel carries req_cfu[3:0] (the CF_ID; named req_cxu in FPGA-IPUG-02302-1.0), req_func[3:0], req_state[2:0], req_insn[31:0], and two 32-bit operands req_data0 and req_data1; the response channel returns resp_data[31:0] and resp_status[2:0]. Each channel uses a valid/ready handshake.

```

module sha3_cxu (sys_clk, sys_rst_n, req_valid, req_ready, req_func,
    req_data0, req_data1, req_state, req_cfu, req_insn, resp_valid,
    resp_ready, resp_data, resp_status);
    input sys_clk;
    input sys_rst_n;
    input req_valid;
    output req_ready;
    input [3:0]req_func;
    input [31:0]req_data0;
    input [31:0]req_data1;
    input [2:0]req_state;
    input [3:0]req_cfu;        // CF_ID; called req_cxu in FPGA-IPUG-02302-1.0
    input [31:0]req_insn;
    output resp_valid;
    input resp_ready;
    output [31:0]resp_data;
    output [2:0]resp_status;
endmodule
// dsp_acc_cxu has an identical port list

```

Figure 5.4. CXU Module Interface (sha3_cxu and dsp_acc_cxu)

5.5. Selecting the Active CXU

Because the two accelerators share the custom-instruction opcodes, firmware selects which CXU is active before issuing CXU instructions by writing the `mcx_selector` CSR (0xBC0). The helper function `w_mcxu_selector()` performs this write. The reference firmware defines two selector values: `CXU_PORT0` (0x20000000) selects `cxu_id` 0, the DSP accelerator, and `CXU_PORT1` (0x20000001) selects `cxu_id` 1, the SHA-3 accelerator. Both values set the `mcx_selector` version field to 1 (bit 29) with `state_id` 0; the low byte holds the `cxu_id`. Note that the SHA-3 accelerator is on the higher-numbered port. (The firmware helper is spelled `w_mcxu_selector`; the CSR itself is named `mcx_selector` in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302).

```
#define CXU_PORT0 0x20000000 // version=1, state_id=0, cxu_id=0 -> DSP accelerator
#define CXU_PORT1 0x20000001 // version=1, state_id=0, cxu_id=1 -> SHA-3 accelerator

// cxu.h: write the mcx_selector CSR (0xBC0) to choose the active CXU port
static inline void w_mcxu_selector(uint32_t x)
{
    asm volatile("csrw 0xbc0, %0" : : "r" (x));
}

// main.c: switch to the SHA-3 CXU, then call the accelerated routine
w_mcxu_selector(CXU_PORT1);
SHA3_cxu(mode, msg_data, MSG_LEN, dgst_data);
...
// switch to the DSP CXU for CNN convolution
w_mcxu_selector(CXU_PORT0);
cnn_convolution_cxu(OP_TIMES);
```

Figure 5.5. Selecting the Active CXU (`cxu.h` and `main.c`)

5.6. SHA-3 CXU Firmware Walkthrough

The SHA-3 CXU firmware (`sha3_cxu.c`) maps the Keccak-f[1600] permutation onto CXU register-file operations. The 5x5 array of 64-bit lanes that forms the Keccak state is held inside the CXU; the firmware loads and stores lanes and drives each Keccak step with custom instructions. The macros used (`opcode_l`, `opcode_flex25`, `opcode_flex25_S`) are defined in `cxu.h` and described in the [Preprocessor Macros](#) section.

5.6.1. Keccak State Load and Store

`store_A()` writes all 25 lanes into the CXU register file with `opcode_flex25_S` (CUSTOM2, func3 = 0), one instruction per lane; each 64-bit lane is split into high and low 32-bit halves by `data_H()` and `data_L()`. `load_A()` reads the lanes back with `opcode_l` (CUSTOM1): func3 = 0 returns the high word and func3 = 1 the low word, with the lane address in the immediate.

```
static uint32_t data_H(uint64_t x) { return (x >> 32); }
static uint32_t data_L(uint64_t x) { return (x & 0xffffffff); }

// store one lane into the CXU (CUSTOM2, func3 = 0), addressed by the rs3 field
opcode_flex25_S(CUSTOM2, 0, 0, data_H(A[0][0]), data_L(A[0][0]), 0b00, 0b000000);
...

// read one lane back (CUSTOM1): func3 = 0 -> high word, func3 = 1 -> low word
A[0][0] = ((uint64_t)opcode_l(CUSTOM1, 0, 0, 0b000000000000) << 32)
          | opcode_l(CUSTOM1, 1, 0, 0b000000000000);
```

Figure 5.6. Keccak State Load and Store (`sha3_cxu.c`)

5.6.2. Keccak-f[1600] Rounds

SHA3_ProcessBlock_cxu() runs the 24 Keccak rounds. Each round issues the five step functions in order: theta, rho, pi, chi, and iota.

```
for (t = 0; t < ctx->nr; t++) /* ctx->nr = 24 rounds */
{
    theta_cxu(); /* column parity and D[x] mixing */
    rho_cxu(); /* per-lane rotations */
    pi_cxu(); /* lane permutation */
    chi_cxu(); /* non-linear step */
    iota_cxu(t); /* XOR round constant into A[0][0] */
}
```

Figure 5.7. Keccak-f[1600] Rounds (sha3_cxu.c)

Each step function issues a sequence of flex-type instructions (opcode_flex25) whose func3 field selects the operation. The following table lists the func3 encodings as used by the SHA-3 firmware.

Table 5.2. opcode_flex25 func3 Operations Used by the SHA-3 Firmware

func3	Operation	Use in Keccak-f
2	Rotate left (ROTL)	theta: ROTL(C[(x+1)%5], 1); rho lane rotations
3	XOR	theta: column parity C[x] and A[y][x] ^ D[x]; chi final XOR
4	AND	chi: (~B[x+1]) & B[x+2]
5	Move / copy	pi: lane permutation
6	NOT (complement)	chi: complement of B[x+1]

The store, absorb, and load instructions use the remaining func3 values: opcode_flex25_S func3 = 0 stores a lane and func3 = 1 XORs a round constant or absorbs a message-schedule word; opcode_I func3 = 0/1 reads a lane high/low word. Because sha3_cxu is delivered as encrypted soft IP, these encodings are taken from the firmware's use of the CXU (cxu.h and sha3_cxu.c), which is the accelerator programming interface.

5.6.3. Software and CXU Benchmark

main.c benchmarks a software implementation against the CXU implementation of each workload. For SHA-3 it hashes a 2048-byte message (MSG_LEN) in the default SHA3_256 mode, timing the software SHA3() and the CXU SHA3_cxu() with the general-purpose timer at 0x40081000 (free-running counter at offset 0x1c). The CXU port is selected before the accelerated pass. The CNN convolution path is timed similarly using sys_ms(). The elapsed times are printed in a summary at the end of main(); the CXU passes complete in fewer timer ticks than the software passes.

```
gp_timer_quick_init(0x40081000);
...
val1 = *((volatile uint32_t *) (0x40081000 + 0x1c));
SHA3(mode, msg_data, MSG_LEN, dgst_data); /* pure software */
val2 = *((volatile uint32_t *) (0x40081000 + 0x1c));
...
w_mcxu_selector(CXU_PORT1); /* select SHA-3 CXU */
val1 = *((volatile uint32_t *) (0x40081000 + 0x1c));
SHA3_cxu(mode, msg_data, MSG_LEN, dgst_data); /* CXU accelerated */
val2 = *((volatile uint32_t *) (0x40081000 + 0x1c));
...
printf("Executing %s ... pure software code cost %d.%d ms.\n", ...);
printf("Executing %s ... CXU extension code cost %d.%d ms.\n", ...);
```

Figure 5.8. Software and CXU-Accelerated SHA-3 Timing (main.c)

5.7. Legacy Example: Simple Endianness Conversion

This subsection retains the original single-CXU endianness example from an earlier revision, kept for reference; it predates the two-CXU SHA3 and DSP design described above.

The following design example demonstrates one way that a minimal CFU design could be implemented. The design performs a trivial endianness swap. Given a 32-bit input, the CFU accelerator in this example performs byte swapping in a single instruction.

This design enables one CFU-L2 port on a RISC-V RX and exports it from the Propel Builder software graphical layer to a higher level register transfer level (RTL) module in the hierarchy. Within the RTL module, the CFU-L2 interface is connected to the custom endianness swapping CFU module, which is also implemented in RTL (as opposed to being packaged for instantiation within the Propel Builder software).

5.7.1. Propel Builder Software

Within an existing Propel Builder RISC-V design, enable one CFU port by enabling the **Enable cfu port** checkbox in the RISC-V RX configuration GUI and setting **Number of CFU** to **1**, as shown in [Figure 5.9](#).

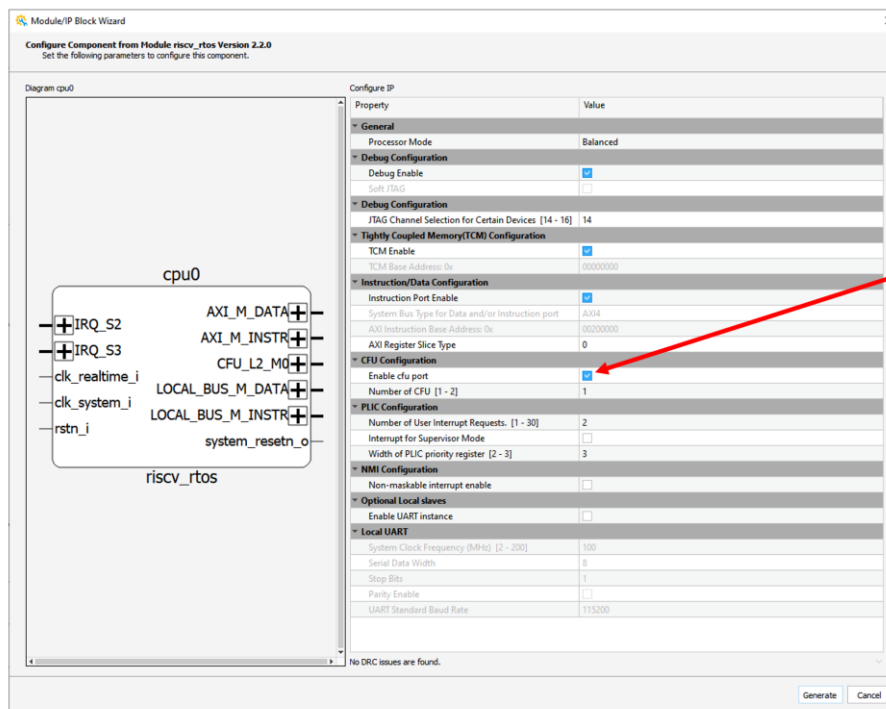


Figure 5.9. Enable CFU Port on RISC-V RX

After the CFU port has been enabled, it can be exported to a higher level of the hierarchy by right-clicking on the CFU_L2_M0 port on the RISC-V and selecting **Export**. The system clock and system reset signals should also be exported by creating output ports and connecting them to the respective signals. To create output ports, from the **Edit** menu, select **Create Port**.

After the CFU port, system clock, and system reset have been exported (see [Figure 5.10](#)), validate, generate the design, and run the Lattice Radiant software from the Propel Builder software. For more information on how to run the Radiant software from the Propel Builder software, refer to the [Lattice Propel 2026.1 Builder User Guide \(FPGA-UG-02185\)](#).

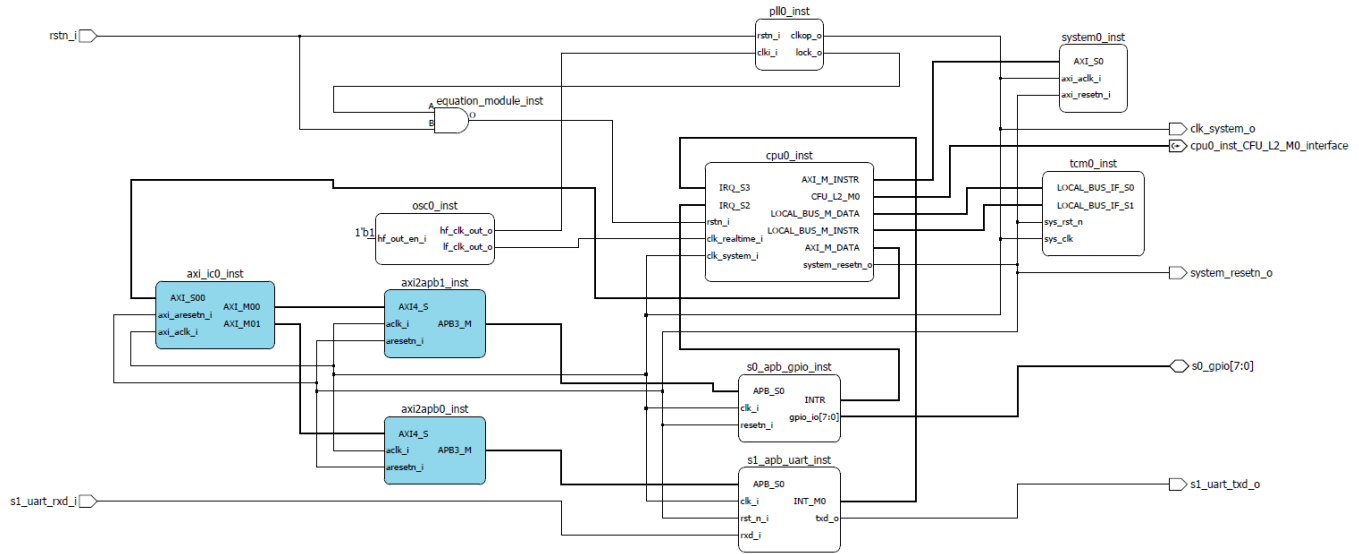


Figure 5.10. RISC-V RX Design Exporting CFU-L2, Clock, and Reset

5.7.2. Top-Level RTL

Figure 5.11 shows a Verilog implementation example of a top-level module that instantiates and connects the RISC-V RX SoC design from the Propel Builder software to the endianness swapping CFU module.

```
module SoC_LFCPNX_Eval_RV32IMC_RX_CFU_endianness_top (
    input          rstn_i,
    input          s1_uart_rxd_i,
    output         s1_uart_txd_o,
    output [7:0]   leds_o
);

    wire           clk_system;
    wire           system_resetrn;

    wire [7:0]     s0_gpio;

    wire [3:0]     req_cfu;
    wire [31:0]    req_data0;
    wire [31:0]    req_data1;
    wire [2:0]     req_func;
    wire [31:0]    req_insn;
    wire [2:0]     req_state;
    wire [31:0]    resp_data;
    wire           req_ready;
    wire           req_valid;
    wire           resp_ready;
    wire           resp_valid;

    assign leds_o = s0_gpio;

    SoC_LFCPNX_Eval_RV32IMC_RX_CFU_endianness soc0(
        .clk_system_o      (clk_system),
        .system_resetrn_o  (system_resetrn),
        .rstn_i            (rstn_i),
        .s1_uart_rxd_i     (s1_uart_rxd_i),
        .s1_uart_txd_o     (s1_uart_txd_o),
        .req_cfu            (req_cfu),
        .req_data0         (req_data0),
        .req_data1         (req_data1),
        .req_func          (req_func),
        .req_insn          (req_insn),
        .req_state         (req_state),
        .resp_data         (resp_data),
        .req_ready         (req_ready),
        .req_valid         (req_valid),
        .resp_ready        (resp_ready),
        .resp_valid        (resp_valid)
    );
endmodule
```

```

        .s1_uart_rxd_i      (s1_uart_rxd_i),
        .s1_uart_txd_o      (s1_uart_txd_o),
        .s0_gpio            (s0_gpio),
        .cpu0_inst_CFU_L2_M0_interface_req_cfu_0_portbus    (req_cfu),
        .cpu0_inst_CFU_L2_M0_interface_req_data0_0_portbus  (req_data0),
        .cpu0_inst_CFU_L2_M0_interface_req_data1_0_portbus  (req_data1),
        .cpu0_inst_CFU_L2_M0_interface_req_func_0_portbus   (req_func),
        .cpu0_inst_CFU_L2_M0_interface_req_insn_0_portbus   (req_insn),
        .cpu0_inst_CFU_L2_M0_interface_req_state_0_portbus  (req_state),
        .cpu0_inst_CFU_L2_M0_interface_resp_data_0_portbus  (resp_data),
        .cpu0_inst_CFU_L2_M0_interface_req_ready_0_port     (req_ready),
        .cpu0_inst_CFU_L2_M0_interface_req_valid_0_port     (req_valid),
        .cpu0_inst_CFU_L2_M0_interface_resp_ready_0_port    (resp_ready),
        .cpu0_inst_CFU_L2_M0_interface_resp_valid_0_port    (resp_valid)
    );

    swap_endianness cfu_0 (
        .clk_i            (clk_system),
        .rstn_i           (system_resets),
        .req_cfu_i         (req_cfu),
        .req_data0_i       (req_data0),
        .req_data1_i       (req_data1),
        .req_func_i        (req_func),
        .req_insn_i        (req_insn),
        .req_state_i       (req_state),
        .resp_data_o       (resp_data),
        .req_ready_o       (req_ready),
        .req_valid_i       (req_valid),
        .resp_ready_i      (resp_ready),
        .resp_valid_o      (resp_valid)
    );
endmodule

```

Figure 5.11 Top-Level RTL

5.7.3. Endianness Swapping CFU Module RTL

Figure 5.12 shows the RTL example for the endianness swapping module.

```

module swap_endianness (
    input    clk_i,
    input    rstn_i,

    /* CFU-L2 interface signals */
    input  [3:0] req_cfu_i,
    input  [31:0] req_data0_i,
    input  [31:0] req_data1_i,
    input  [2:0] req_func_i,
    input  [31:0] req_insn_i,
    input  [2:0] req_state_i,
    output [31:0] resp_data_o,
    output      req_ready_o,
    input      req_valid_i,
    input      resp_ready_i,
    output      resp_valid_o,
    output  [2:0] resp_status_o
)

```

```

);

localparam CFU_OK = 3'b000;

reg          req_ready_r;

reg  [31:0]  resp_data_r;
reg          resp_valid_r;

assign resp_data_o    = resp_data_r;
assign resp_valid_o   = resp_valid_r;
assign req_ready_o    = resp_ready_i;
assign resp_status_o  = CFU_OK;          /* always send CFU_OK status */

always @(posedge clk_i or negedge rstn_i)
begin
    if (rstn_i == 1'b0)
    begin
        resp_data_r <= 32'h00000000;
        resp_valid_r <= 1'b0;
    end
    else
    begin
        if (req_valid_i)                  /* if a valid CFU request from the CPU is present
*/
            begin
                resp_data_r <= { req_data0_i[7:0], /* swap bytes on data0 input
*/
                                req_data0_i[15:8],
                                req_data0_i[23:16],
                                req_data0_i[31:24] };
                resp_valid_r <= 1'b1;          /* signal response is valid on next cycle
*/
            end
            else if (resp_ready_i)          /* only after CPU has acknowledged the
response...*/
            begin
                resp_valid_r <= 1'b0;        /* ...do we deassert response valid
*/
            end
        end
    end
end
endmodule

```

Figure 5.12. Endianness Swapping CFU Module

5.7.4. Example Firmware

Figure 5.13 shows the firmware that issues requests and reads the response from the endianness swapping CFU module.

```
static void endianness_test()
{
    unsigned int input_arg = 0x01234567;

    unsigned int cfu_en = 1;
    unsigned int state = 0;
    unsigned int cfu_id = 0;
    unsigned int config = (cfu_en << 31) | (state << 16) | (cfu_id << 0);

    unsigned int result;
    unsigned int status;

    // setup mcfu_selector CSR
    // use CSR write pseudo instruction to write mcfu_selector
    __asm__ volatile ("csrw 0xBC0, %[rs]"           // mcfu_selector CSR location = 0xBC0
                     :                               // write operation so OutputOperands is an empty list
                     : [rs]"r"(config)             // write the value of "config" C variable to the CSR
                     );

    // issue instruction to CFU
    // I-type Instruction:
    // +-----+-----+-----+-----+-----+
    // |   imm12   | rs1 | func3 | rd | opcode |
    // +-----+-----+-----+-----+-----+
    // 31         20   15      12   7         0
    //
    // .insn i opcode, func3, rd, rs1, imm12
    __asm__ volatile (".insn i 0x2b, 0, %[rd], %[rs1], 0"
                     : [rd]"=r"(result)
                     : [rs1]"r"(input_arg)
                     );

    // read cfu_status CSR
    __asm__ volatile ("csrr %[stat], 0x801"         // cfu_status CSR location=0x801
                     : [stat]"=r"(status)          // read result stored in "status"
                     );

    printf("Swap Result (0x%02x)-   In: 0x%08x      Out: 0x%08x\r\n", status, input_arg, result);

    return;
}
```

Figure 5.13. Endianness Swapping Example Firmware

References

- [Lattice Propel 2026.1 Builder User Guide \(FPGA-UG-02185\)](#)
- [Lattice Propel 2026.1 SDK User Guide \(FPGA-UG-02186\)](#)
- [RISC-V RX CPU IP User Guide \(FPGA-IPUG-02302\)](#)
- [RISC-V Composable Custom Extensions Specification \(0.91.230803\)](#)
- [RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA | Five EmbedDev \(five-embeddev.com\)](#)
- [RISC-V Instruction Set Manual, Volume II: Privileged Architecture | Five EmbedDev \(five-embeddev.com\)](#)
- [RISC-V-Directives \(sourceware.org\)](#)
- [Extended Asm \(Using the GNU Compiler Collection \(GCC\)\)](#)
- [RISC-V RX CPU IP Core](#) web page
- [Lattice Propel Design Environment](#) web page
- [Lattice Insights](#) web page for Lattice Semiconductor training courses 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/en/Support/AnswerDatabase.

Revision History

Revision 1.2, September 2026

Section	Change Summary
All	Updated the CFU terminology to CXU (Custom Function Unit to Composable Extension Unit) and CFU-LI to CXU-LI.
Abbreviations in This Document	Added the following abbreviations—ANSI, CNN, CF, CX, CXE, CXU, CXU-LI, DSP, GPIO, IP, LED, MSB, and UART.
Introduction	<ul style="list-style-type: none"> Updated the introduction in this section and added a new note. Updated the Nomenclature section.
Overview	Updated the content in this section and replaced Figure 2.1. Multi-CXU System .
CXU-LI: Interface Between RISC-V and CXU	<ul style="list-style-type: none"> Updated the introductory paragraph in this section. Updated Table 3.1. CXU-LI Port Definitions. Updated the CXU Feature Levels section. Updated the CXU-L2 in the Lattice Propel Builder Software section.
CXU Software Model	<ul style="list-style-type: none"> Updated the CSRs section. Updated the Instructions section. Updated the I-type Encoding section. Updated the Assembly Language section. Updated the I-Type Instruction Inline Assembly Example section. Updated the CXU CSR Access Inline Assembly section. Updated the SHA-3 Template Macros section.
Design Examples	Replaced this section with two sections about CXU SHA-3 and DSP reference design built using the Propel Builder scalable template flow (Lattice Radiant 2026.1, RISC-V RX CPU IP v2.8.1, CertusPro-NX LFCPNX-100). The previous endianness example is retained as a legacy reference.
References	Updated the link to the RISC-V RX CPU IP User Guide (FPGA-IPUG-02302) and the Lattice Propel Builder 2026.1 documentation.

Revision 1.1, March 2024

Section	Change Summary
Overview	<p>Updated the following sentences in this section:</p> <ul style="list-style-type: none"> <i>The specification allocates three of the opcodes that the RISC-V instruction set architecture (ISA) reserves for custom instructions</i> <i>A request will generally include two operands: either the contents of two of the registers in the CPU's register file or a single register from the register file and an immediate value.</i> <i>The Composable Custom Extensions Specification also assigns a small subset of the Control and Status Registers (CSRs) that are reserved by the RISC-V ISA for the purposes of adding custom features.</i>
CXU-LI: Interface between RISC-V and CXU	<p>Updated the following signal names in Table 3.1. CFU-LI Port Definitions:</p> <ul style="list-style-type: none"> clk rst req_cfu
CXU Software Model	<ul style="list-style-type: none"> Updated the following sentence in this section: <i>The RISC-V ISA reserves a set of instruction opcodes as well as address blocks within control status register (CSR) space to enable the addition of custom instructions and functionality.</i> Updated the 4.2 Instructions section. Added the 4.2.4 Software Model and the CFU-LI section. Updated the 4.3.1 Assembly Language section. Added the 4.3.1.1 .insn Assembly Directive section. Added the 4.3.1.2 .word Assembly Directive section.

Section	Change Summary
	<ul style="list-style-type: none">• Updated section title <i>4.3.2.1 asm Keyword</i> to <i>4.3.2.1 Inline Assembly</i>.• Updated the Function ID 0 to 3 in the 4.3.2.2 I-Type Instruction Inline Assembly Example section.• Updated the code in the 4.3.2.4 Flex-Type Instruction Inline Assembly Example section.• Updated the code in the 4.3.2.5 CFU CSR Access Inline Assembly section.• Added 4.3.3 Preprocessor Macros section.• Updated the code in Figure 5.4. Endianness Swapping CFU Module.• Updated the code in Figure 5.5. Endianness Swapping Example Firmware

Revision 1.0, December 2023

Section	Change Summary
All	Initial release.



www.latticesemi.com