Composable Custom Extensions on Lattice RISC-V RX User Guide

Application Note

FPGA-AN-02075-1.1

March 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE 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.

Acronym	Definition
ABI	Application Binary Interface
CFU	Custom Function Unit
CFU-LI	Custom Function Unit Logical Interface
CPU	Central Processing Unit
CSR	Control and Status Register
CX	Composable Extension
CXU	Composable Extension Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
ISA	Instruction Set Architecture
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

1. Introduction

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

The purpose of this application note is to provide an introduction on how to implement a simple CFU, connect it to the RISC-V RX core, and write firmware that executes CFU-specific instructions.

1.1. Naming Conventions

1.1.1. Nomenclature

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

Note: The RISC-V Composable Custom Extensions Specification had recently been updated to change key terminology custom function unit (CFU) to composable extension unit (CXU). However, the RISC-V Composable Custom Extensions Specification is still in the draft stages and is subject to change. This document will continue to use the older terminology—CFU—until the newer term is widely adopted by the community. For more information, refer to the latest RISC-V RX CPU IP User Guide (FPGA-IPUG-02230) and the RISC-V Composable Custom Extensions Specification.

1.1.2. Signal Names

- _*n* are active low (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 CFU logical interface (CFU-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, a CFU, and the CFU-LI that connects the two.

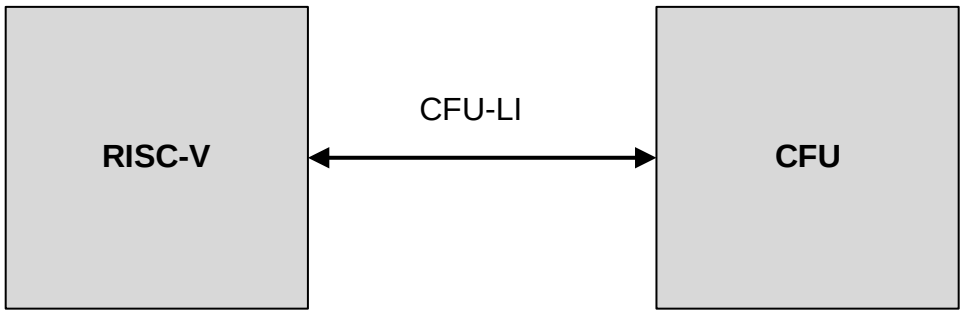

Figure 2.1. Simple CFU-Based System

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 CFU-LI to the targeted CFU.

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. The CFU-LI can use handshaking similar to AXI4 for both request and response transactions and this allows a CFU to support fixed or variable length processing delays. Once the CFU has processed a request, it responds with status information and an optional result. The result is stored in the RISC-V's 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 purposes of adding custom features. These CSRs are used to configure the CFU-LI, to select the current target CFU, and to select the CFU's state context, if the implemented function is stateful and multiple threads are using it. CFU status information is accumulated in one of these CSRs and can be read back or cleared via standard RISC-V CSR read/write instructions.

Note: The Lattice Propel™ Builder design environment does not provide CFU switching infrastructure and does not support the composable custom extensions feature, *IStateContext*.

3. CFU-LI: Interface Between RISC-V and CFU

The CFU-LI is an interconnect and protocol by which the RISC-V issues requests to the CFU and the CFU communicates responses.

3.1. Signal Definitions

The signals defined in the CFU specification are repeated in Table 3.1.

Table 3.1. CFU-LI Port Definitions

Signal Name	Direction	Width Parameter	Width	Level	Description
clk	system -> CFU	—	1	1+	CPU core clock
rst	system -> CFU	—	1	1+	System reset
clk_en	system -> CFU	—	1	1+	Clock enable
req_valid	CPU -> CFU	—	1		Request Valid: Handshaking signal indicating that the CPU is sending a request.
req_ready	CFU -> CPU	—	1	2+	Request Ready: Handshaking signal indicating that CFU is ready to receive a new request. Required for variable latency CFU types.
req_id	CPU -> CFU	CFU_REQ_ID_W	N/A	3	Request ID: To track outstanding requests with reordering (Feature Level 3) type CFUs. Not supported by RISC-V RX.
req_cfu	CPU -> CFU	CFU_CFU_ID_W	4	All	Request CFU ID: To identify which CFU is being requested in a multiple CFU design.
req_state	CPU -> CFU	CFU_STATE_ID_W	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 CFU.
req_func	CPU -> CFU	CFU_FUNC_ID_W	3	All	Request Function ID: Selects which function to perform within the specified CFU (For example, load state vs. accumulate).
req_insn	CPU -> CFU	CFU_INSN_W	32	2+	Request Raw Instruction: The entire 32-bit instruction whose execution generated the request.
req_data0	CPU -> CFU	CFU_DATA_W	32	All	Request Operand Data 0: The first operand of the requested CFU operation.
req_data1	CPU -> CFU	CFU_DATA_W	32	All	Request Operand Data 1: The second operand of the requested CFU operation.
resp_valid	CFU -> CPU	—	1	1+	Response Valid: Handshaking signal from the CFU indicating that the CFU has completed the requested operation and that the result is available on resp_data.
resp_ready	CPU -> CFU	—	1	2+	Response Ready: Handshaking signal from the CPU indicating that it is ready to receive the CFU response.
resp_id	CFU -> CPU	CFU_REQ_ID_W	N/A	3	Response ID: To track outstanding requests with reordering (Feature level 3) type CFUs. Not supported by RISC-V RX.
resp_status	CFU -> CPU	CFU_STATUS_W	6	All	Response Status: Carries a success or error code from the CFU in response to a request.
resp_data	CFU -> CPU	CFU_DATA_W	32	All	Response Data: The result of the CFU operation. Written to a register within the register file specified by the rd field of the instruction that initiated the request.

3.2. CFU Feature Levels

The CFU Specification defines four feature levels (0-3), as shown in Table 3.2. However, the RISC-V RX is configured to support feature level 2 only. Consequently, the CFU ports on the RISC-V RX are of type *CFU-L2*.

Table 3.2. CFU Feature Levels

Feature Level	CFU Type	RISC-V RX Support
0	Combinatorial	Requires an adapter
1	Fixed latency	Requires an adapter
2	Variable latency	Supported
3	Reordering	Not supported

Feature levels 0 and 1 can be supported by driving the level 2 inputs to the CPU core to the appropriate constant logic levels. Feature level 3 is not supported.

3.3. CFU-L2 in the Lattice Propel Builder Software

When a RISC-V RX is instantiated within the Lattice Propel Builder software, one or two CFU ports can be exported by enabling the checkbox for the **Enable cfu port** option and selecting the number of CFU ports, as shown in Figure 3.1.

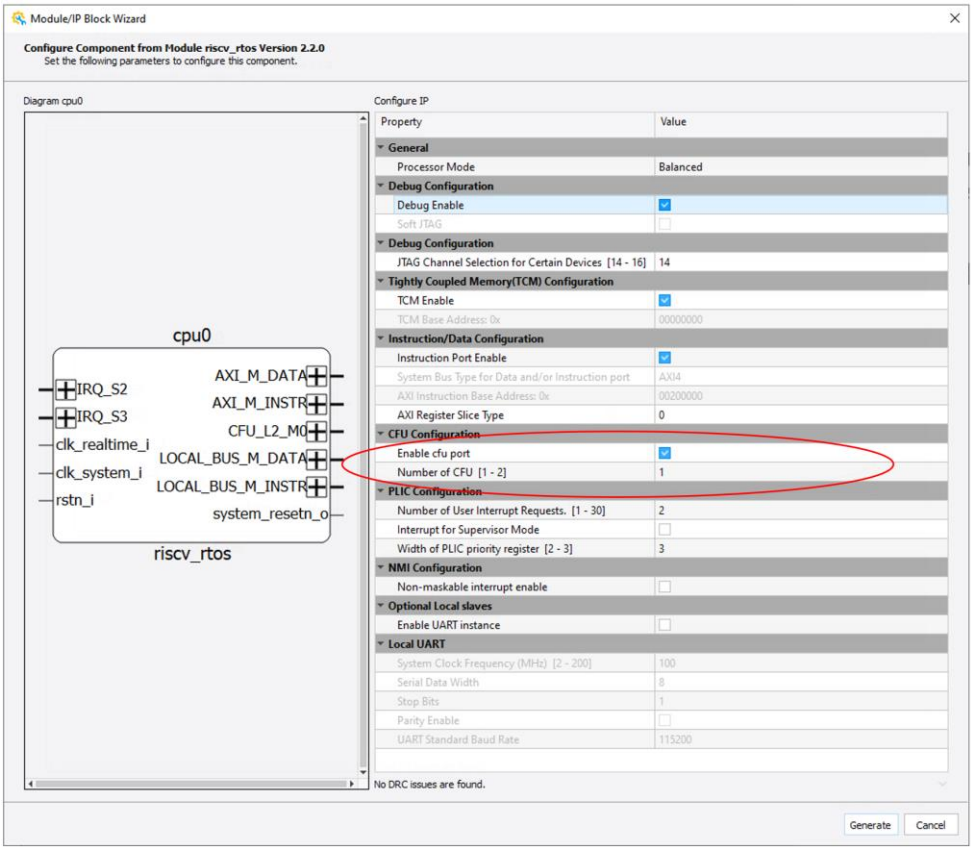

Figure 3.1. RISC-V RX Configuration GUI

Once a CFU-enabled RISC-V RX is present in a Propel Builder design, it can be connected to your custom logic by exporting the CFU-L2 port to a higher level in the hierarchy, as shown in Figure 3.2.

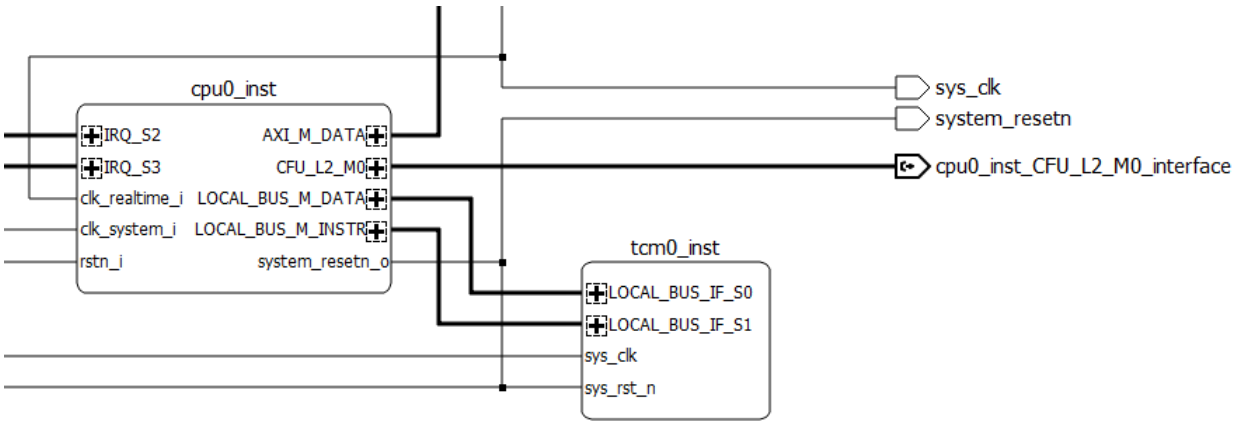

Figure 3.2. CFU-LI Exported to Higher Level of Hierarchy

4. CFU Software Model

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. The CFU specification assigns functionality to subsets of both resources.

4.1. CSRs

The CFU specification defines the following four new CSRs:

- mcfu_selector: Selects the active CFU and the state context within that CFU.
- cfu_status: Accumulates error flags from CFU.
- mcfu_selector_table: **Not currently implemented in RISC-V RX.** Holds the base address of the *CFU selector table* used to allow unprivileged code to modify the mcfu_selector.
- cfu_selector_index: **Not currently implemented in RISC-V RX.** User mode accessible CSR that allows unprivileged code to cause the mcfu_selector to be written with the CFU ID and state context stored at the specified index in the *CFU selector table*.

4.1.1. mcfu_selector CSR

The mcfu_selector CSR is used to select the active CFU and its state context.

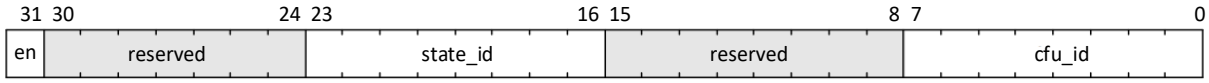

Figure 4.1. mcfu_selector CSR (0xBC0)

en: CFU-L2 enable. Enables custom interface multiplexing when set.

state_id: Selects the current state ID. The **lower three bits** of this field will be driven on the *req_state* port of the selected CFU-L2 port during a CFU request.

cfu_id: Selects the current CFU ID. The **lower four bits** of this field will be driven on the *req_cfu* port of the RISC-V's CFU-L2 port during a CFU request. If the RISC-V RX is configured to export two CFU-L2 interfaces, the **least significant bit** of this field will control which port (Port 0 or Port 1) will send subsequent requests.

4.1.2. cfu_status CSR

The cfu_status CSR accumulates error flags returned by the CFU(s). Error flag bits can be cleared by writing zeros to the CSR.

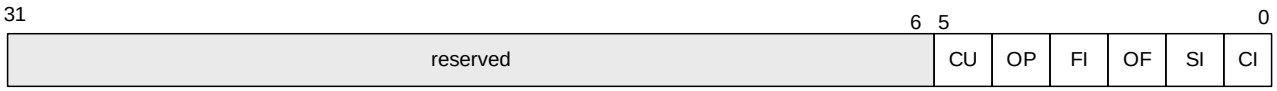

Figure 4.2. cfu_status CSR (0x801)

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

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 4.1.

Table 4.1. RISC-V ISA Opcodes (inst[1:0] = 11)

inst [4:2] inst [6:5]	000	001	010	011	100	101	110	111 (> 32b)
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 instructions:

- CUSTOM0 for the R-type CFU instruction
- CUSTOM1 for the I-type CFU instruction
- CUSTOM2 for the flex CFU instruction

4.2.1. R-type Encoding

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

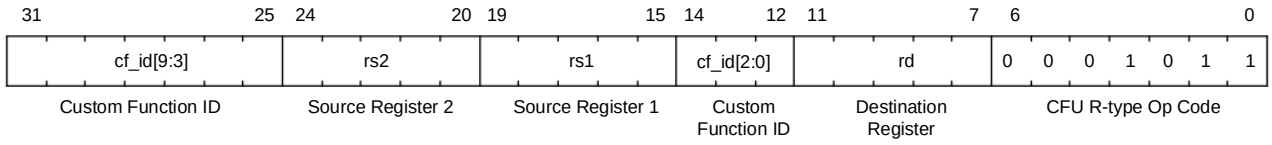

Figure 4.3. CFU R-type Instruction

4.2.2. I-type Encoding

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

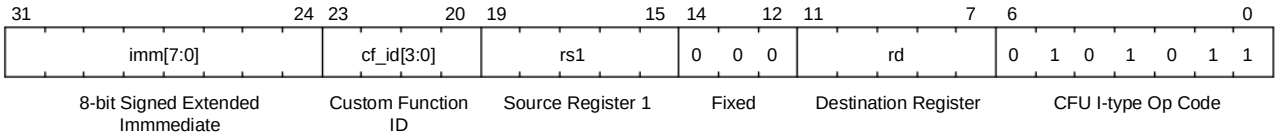

Figure 4.4. CFU I-type Instruction

4.2.3. Flex-type Encoding

The CFU Specification's flex-type encoding is used to initiate operations that do not require a response. For example, 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 4.5 and an alternate form in Figure 4.6.

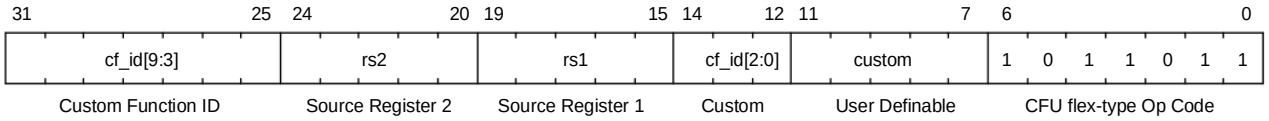

Figure 4.5. CFU flex-type Instruction

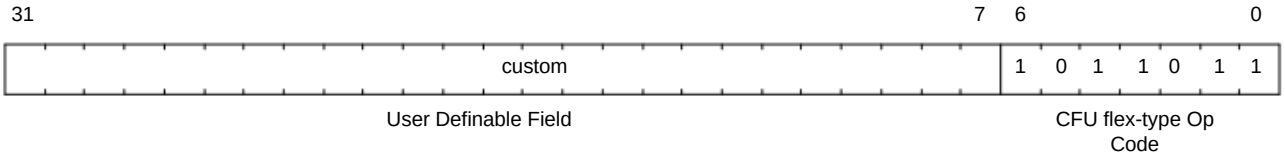

Figure 4.6. CFU flex-type Instruction Alternate Encoding

The primary purpose of the flex-type encoding is to provide a mechanism to alter the currently selected state context of the targeted CFU. The custom fields of both flex-type forms are available to the connected CFU(s) via the CFU-LI's raw instruction bus—*req_insn*—and can be used as you see fit.

4.2.4. Software Model and the CFU-LI

Figure 4.7 shows the relationship between the instructions and CSRs of the CPU and the CFU-LI.

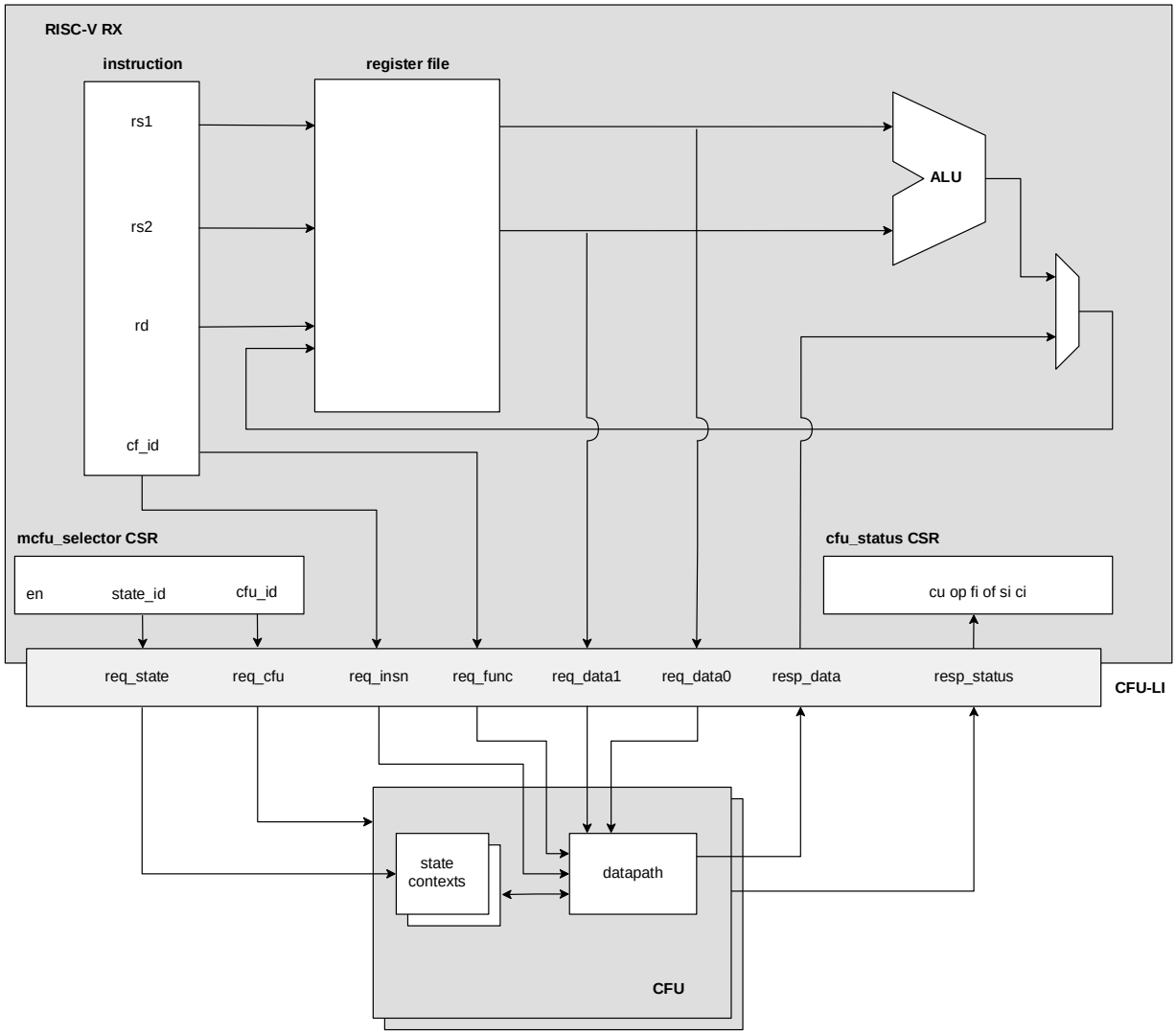

Figure 4.7. Relationship between Instructions and CSRs of the CPU and the CFU-LI

The *req_state* and *req_cfu* busses originate in the *mcfu_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 that are indexed by the *rs1* and *rs2* fields of the R-type instruction, respectively (for the I-type instruction, *req_data1* is sourced by the *imm* field of the instruction).

The 32-bit result of the CFU 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. Any errors during the transaction are accumulated in the *cfu_status* CSR.

4.3. Firmware

4.3.1. Assembly Language

The Composable Custom Extensions Specification describes a new assembler instruction, `cfu_reg (cx_reg)`, that enables CFU custom instructions to be initiated. Note that the standard GNU assembler that currently ships with the Propel SDK does not support the `cfu_reg (cx_reg)` instruction.

CFU 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 CFU 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 CFU instruction

cf_id[2:0] – lower 3 bits of the Function ID within the targeted CFU

cf_id[9:3] – upper 7 bits of the Function ID within the targeted CFU

rd – destination register

rs1 – is the first source register

rs2 – is the second source register

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

```
.insn  i   0x2B, 0x0, rd, rs1, (imm << 4 | cf_id)
```

Where:

i – denotes the type of encoding (I-type).

0x2B – is the opcode for the I-type CFU instruction.

0x0 – is the CFU I-type's "fixed" bit field of all zeros.

rd – specifies destination register.

rs1 – specifies the source register.

imm – is the eight-bit sign extended immediate value.

cf_id – is the Function ID (4-bits) within the targeted CFU.

The main version of the CFU flex-type instruction is similar to the R-type, except that the CFU response data is not written back to the CPU's register file. In the place of the destination register field—*rd*—the flex-type instruction has a 5-bit user defined immediate value. So, the CFU flex-type instruction can be written in assembly as:

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

Where:

r – Denotes the type of encoding (R-type).

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

cf_id[2:0] – lower 3 bits of the Custom Function ID.

cf_id[9:3] – upper 7 bits of the Custom Function ID.

custom – 5-bit user defined immediate value
(`.insn` expects a register name. For example, one of x0 through x31).

rs1 – is the first source register.

rs2 – is 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 a simple 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:

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

```
unsigned int arg=10;    // some arbitrary input
unsigned int result;      // variable to store the result of the cfu computation
__asm__ volatile (“.insn i 0x2B, 0, %[rd], %[rs1], (12 << 4 | 0x03)”
      : [rd]”=r”(result)
      : [rs1]”r”(arg)
      );
```

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 umber 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. CFU CSR Access Inline Assembly

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

```
unsigned int cfu_en = 1;
unsigned int cfu_id = 0;
unsigned int state_id = 5;
unsigned int config = (cfu_en << 31) | (state_id << 16) | (cfu_id << 0);

__asm__ volatile ("csrw 0xBC0, %[rs1]" // mcfu_selector CSR address=0xBC0
                  :                    // empty OutputOperands list
                  : [rs1]"r"(config)   // use C variable, config, as input
                  );
```

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

```
unsigned int status;

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

4.3.3. Preprocessor Macros

The C/C++ preprocessor operates via 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 CFU 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:

identifier –	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.
param1, param2,… -	Optional parameters. An argument placed in the position of a given parameter will be used to replace any instance of the parameter’s name within the token-string.
token-string -	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’s 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 a user of the macro would intuitively expect. 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 tertiary 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 simply 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 Propel Builder software 2024.1 and later includes a template that demonstrates the computation of the SHA-3 hash using the CFU feature.

The firmware portion of the template uses macros to simplify the process of writing inline assembly commands that insert the CFU custom instructions. These include macros for the R-type, I-type, and several versions of the flex instructions.

Here, we review the macro for the I-type instruction as it provides a good example of how these macros use the previously covered GNU C/C++ preprocessor and compiler features to add custom instructions to the high-level source code.

The SHA-3 template defines the "opcode_I" macro as follows:

```
#define opcode_I(opcode, func3, func4, rs1, imm)             \
({                                                           \
    register unsigned long result;                           \
    asm volatile(                                            \
        ".word (                                             \
        (" #opcode ") |                                      \
        (regnum_%[result] << 7) |                            \
        (regnum_%[arg1] << 15) |                             \
        ((" #imm ") << 24) |                                 \
        ((" #func3 ") << 12) |                               \
        ((" #func4 ") << 20));\n"                            \
     : [result] "=r" (result)                                \
     : [arg1] "r" (rs1)                                      \
    );                                                       \
    result;                                                  \
})
```

The Composable Custom Extensions Specification defines its I-type instruction as shown in Figure 4.4.

The I-type instruction contains six fields: `imm`, `cf_id[3:0]`, `rs1`, `rd` and a fixed, 3-bit field that must be set to all zeros (other values are reserved for future use). The `opcode_I` macro's parameter list corresponds directly with five of these six values:

```
opcode_I(opcode, func3, func4, 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, func3, and func4) 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, etc.) 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 three custom instruction opcodes: CUSTOM0 for R-type instructions, CUSTOM1 for I-type instructions, and CUSTOM2 for flex instructions.

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

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

```
result = opcode_I(CUSTOM1, 0, 1, input_arg, 213);
```

Where:

`CUSTOM1:`	The RISC-V ISA *custom-1* opcode (0x2B) that is used by the Custom Composable Extensions specification to encode I-type instructions.
`0:`	The value of the `func3` parameter at bit position 12. In the Composable Custom Extensions Specification, this field is reserved and must be set to a value of 0.
`1:`	The value of the `func4` parameter at bit position 20. This is the *cf_id* field in the Composable Custom Extensions Specification. The *cf_id* field must be set with the number of desired CFU function ID.
`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 CFU.
`213:`	The value of the *imm* parameter is placed at bit 24 of the instruction. The 8-bit *imm* field will be sign extended to 32-bits and driven on the *req_data1* bus during the transaction request to the CFU.

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 4.8.

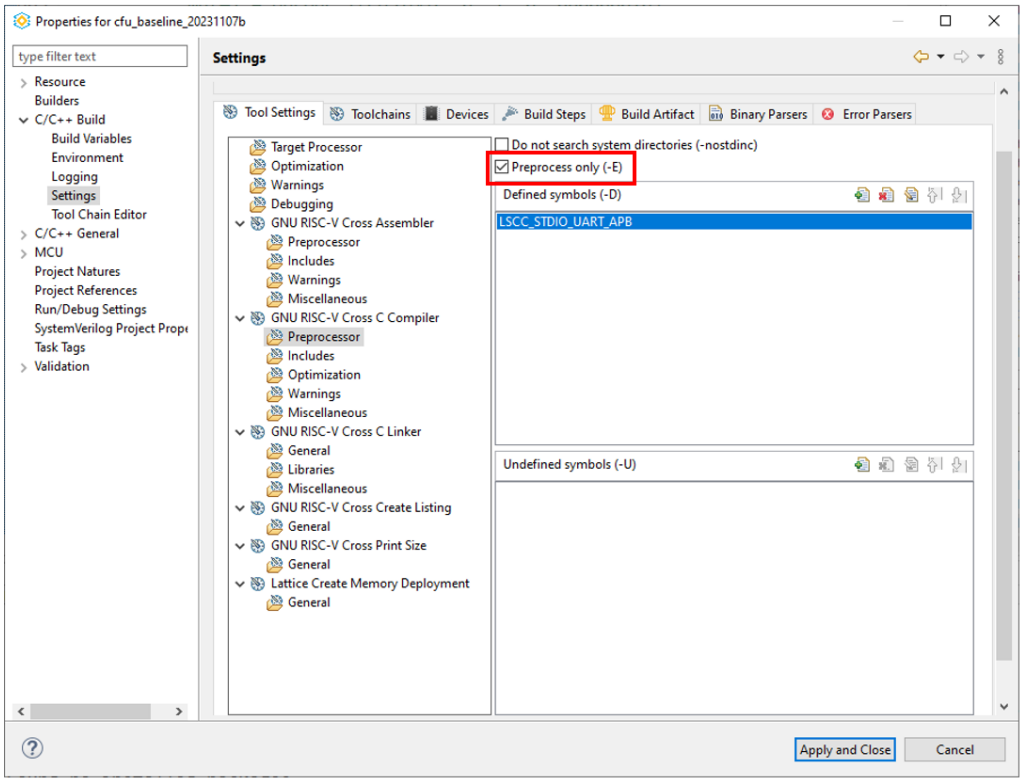

Figure 4.8. Enabling the Preprocess Only Option

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. Simple Endianness Conversion

The following is a simple design example to demonstrate 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's 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).

5.1.1. 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 5.1. .

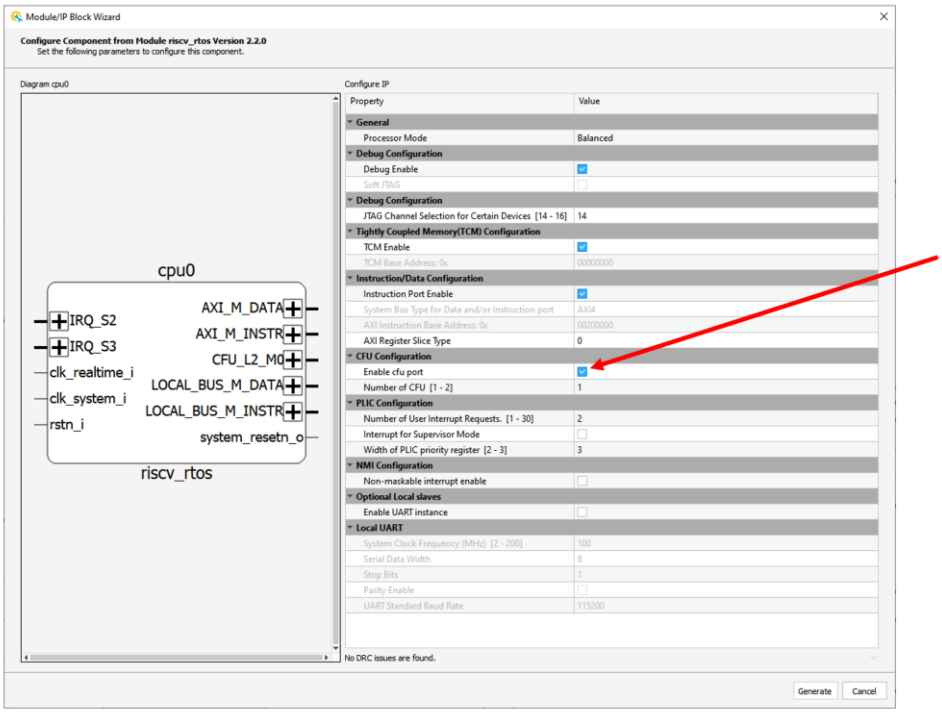

Figure 5.1. Enable CFU Port on RISC-V RX

Once 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**.

Once the CFU port, system clock, and system reset have been exported (see Figure 5.2), 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 Proper Builder software, refer to the Lattice Propel 2023.1 Builder User Guide (FPGA-UG-02185).

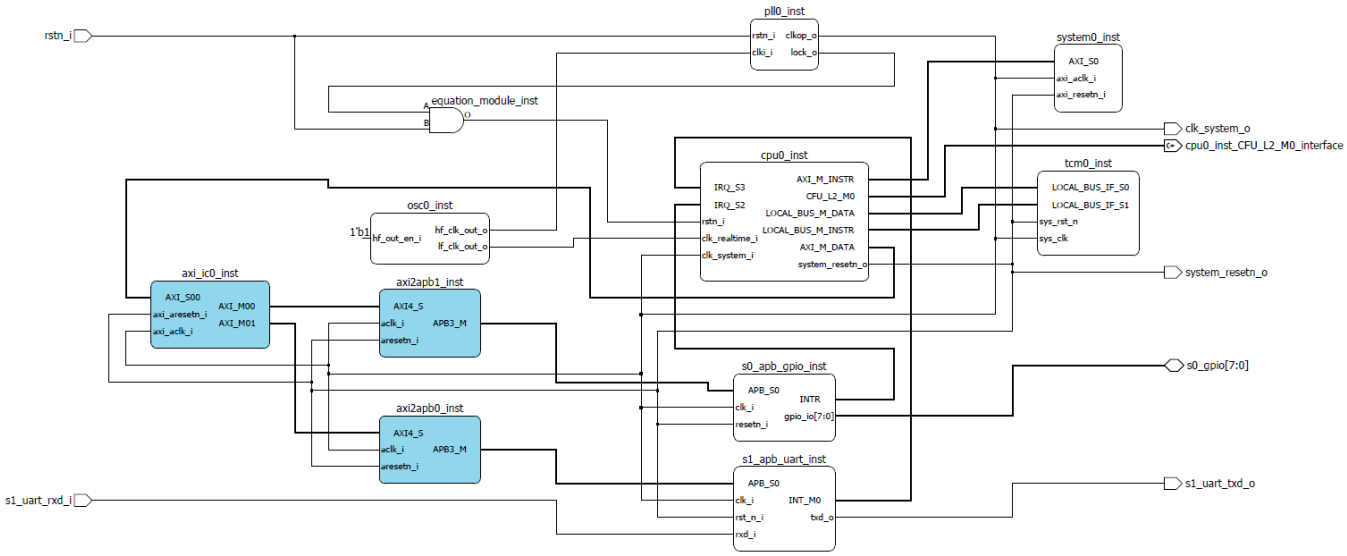

Figure 5.2. RISC-V RX Design Exporting CFU-L2, Clock and Reset

5.1.2. Top-Level RTL

Figure 5.3 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_resetn;

    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_resetn_o                                                (system_resetn),
        .rstn_i                                                         (rstn_i),
        .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_resetn),
        .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 5.3. Top-Level RTL

5.1.3. Endianness Swapping CFU Module RTL

Figure 5.4 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

 endmodule
```

Figure 5.4. Endianness Swapping CFU Module

5.1.4. Example Firmware

Figure 5.5 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 5.5. Endianness Swapping Example Firmware

References

For more information, refer to the following resources:

- Lattice Propel 2023.1 Builder User Guide (FPGA-UG-02185)
- Lattice Propel 2023.1 SDK User Guide (FPGA-UG-02186)
- RISC-V RX CPU IP User Guide (FPGA-IPUG-02230)
- RISC-V Composable Custom Extensions Specification (0.91.230811)
- 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 1.1, March 2024

Section	Change Summary
Overview	Updated the following sentences in this section: • *The specification allocates three of the opcodes that the RISC-V instruction set architecture (ISA) reserves for custom instructions* • *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.* • *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.*
CFU-LI: Interface Between RISC-V and CFU	Updated the following signal names in Table 3.1. CFU-LI Port Definitions: • clk • rst • req_cfu
CFU Software Model	• Updated the following sentence in this section: *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*. • 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. • Updated section title *4.3.2.1 asm Keyword* to *4.3.2.1 Inline Assembly*. • 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