Lattice RISC-V Embedded Design Guidelines

Application Note

FPGA-AN-02072-1.1

February 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 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
AHB-L	Advanced High-performance Bus-Lite
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
BSP	Board Support Package
CDC	Clock Domain Crossing
CLINT	Core Local Interrupter
CPU	Central Processing Unit
CSR	Control and Status Register
CXU-LI	Composable Custom Extension – Local Interface
DDR	Double Data Rate
DMA	Direct Memory Access
DMIPS	Dhrystone MIPS
DQ/DQS	Data/Data Strobe
DRQ	Design Rule Check
EBR	Embedded Block RAM
ECO	Engineering Change Order
ELF	Executable and Linkable Format
FIFO	First In, First Out Buffer
GDB	GNU Debugger
GPIO	General Purpose Input/Output
I2C	Inter-Integrated Circuit
I3C	Improved Inter-Integrated Circuit
IOPMP	Input/Output Physical Memory Protection
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group
LPDDR4	Low Power DDR4
LUT	Look-Up Table
LWIMC	Lightweight Interrupt Merge Controller
MMIO	Memory-Mapped Input/Output
OpenOCD	Open On-Chip Debugger
PCB	Printed Circuit Board
PCLK	Peripheral Clock
PIC	Programmable Interrupt Controller
PLIC	Platform Level Interrupt Controller
PLL	Phase-Locked Loop
PMP	Physical Memory Protection
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer-V
ROM	Read Only Memory
RRID	Request Role ID
RTOS	Real-time Operating System
RVFI	RISC-V Formal Interface

Acronym	Definition
SCLK	SPI Clock
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory.
SGDMA	Scatter-Gather DMA
SPI	Serial Peripheral Interface
SoC	System-on-Chip
TCL	Tool Command Language
TCM	Tightly Coupled Memory
TOR	Top of Range
UART	Universal Asynchronous Receiver/Transmitter.
USB	Universal Serial Bus
XML	eXtensible Markup Language
XiP	Execute-in-Place

1. Introduction

1.1. Overview

Lattice embedded system solutions provide RISC-V processor IP, memory IP, communication IP (UART, I2C, I3C, SPI, and other IPs), along with design software such as Lattice Propel™ Builder, Lattice Propel Software Development Kit (SDK), Lattice Radiant™, and Lattice Diamond™.

- The Lattice Propel Builder is a graphical design tool for SoC hardware design. This software allows you to create a system composed of various IP modules which are provided by Lattice or custom-built. Using the schematic interface, you can easily instantiate and connect modules, while the tool manages address mapping for memory-mapped peripherals. After system assembly, the Lattice Propel Builder generates RTL files and a system environment XML file, which serve as the foundation for embedded software development.
- The Lattice Propel SDK is a software development environment for C-based embedded projects targeting the processor system created in the Lattice Propel Builder. The Lattice Propel SDK generates the Board Support Package (BSP), including processor startup code, device drivers, and platform headers to accelerate application development. The Lattice Propel SDK also provides debugging tools such as OpenOCD and GNU GDB for on-device software debugging.
- The Lattice Radiant and Lattice Diamond software handle FPGA implementation tasks such as synthesis, mapping, place-and-route, and bitstream generation. These tools also support pin assignments, timing constraints, and programming the FPGA device.

1.2. Purpose

This document provides guidance on designing with Lattice embedded solutions and information regarding various design options for Lattice RISC-V processors and IP. The document is organized into sections covering the entire design flow: selecting the RISC-V processor IP, hardware design, software design, and debugging.

1.3. Audience

The intended audience for this document includes embedded system designers and embedded software developers using Lattice FPGA devices. The pre-requisite of this technical guidelines is the knowledge in digital design, FPGAs, and embedded systems.

2. Lattice RISC-V Processors Family

The Lattice Propel software supports four variants of the RISC-V CPU: NANO, SM, MC, and RX. Table 2.1 shows an overview of the models and the basic use cases.

Table 2.1. Lattice RISC-V Use Cases

Model	Class	Supported Device Family	Targeted Software Stack	Typical Application
RX	RTOS Capable	Certus™-N2, Lattice Avant™, MachXO5™-NX, CrossLinkU™-NX, CrossLink™-NX, CertusPro™-NX, Certus-NX	• RTOS (FreeRTOS™, Zephyr) • Bare metal	• Higher performance • Network connected • External memory support
MC	Microcontroller	MachXO4™, Certus-N2, Lattice Avant, MachXO5-NX, CrossLinkU-NX, CrossLink-NX, CertusPro-NX, Certus-NX, MachXO3D™, MachXO3L™, MachXO3LF™, MachXO2™	• Bare metal • RTOS (third party)	• Mid-performance • Microcontroller replacement • External memory support
SM	State Machine	Certus-N2, iCE40 UltraPlus™, Lattice Avant, MachXO5-NX, CrossLinkU-NX, CrossLink-NX, CertusPro-NX, Certus-NX, MachXO3D, MachXO3L, MachXO3LF, MachXO2	Bare metal	• Small footprint • Simple monitoring and configuration applications
NANO	Glue Logic	MachXO4, Certus-N2, iCE40 UltraPlus, Lattice Avant, MachXO5-NX, CrossLinkU-NX, CrossLink-NX, CertusPro-NX, Certus-NX, MachXO3D, MachXO3L, MachXO3LF, MachXO2	Bare metal	• Smallest footprint • Simple monitoring and control applications

The RISC-V NANO is a compact and highly area-optimized implementation of the baseline RV32I instruction set architecture, specifically designed for simple monitoring and control tasks. Acting as a glue logic processor, RISC-V NANO excels in applications where minimal footprint and efficient integration are critical. Optimized for small device families such as MachXO2 and MachXO3, the RISC-V NANO leverages a compact AHB-Lite interconnect and lightweight interrupt merge controller to minimize area usage.

The RISC-V SM is the state machine version and trades performance for reduced area size. This model is for simple monitoring and control tasks. With small size and area efficient AHB-Lite interconnect, this model is suitable for use in smaller device families such as MachXO2 and MachXO3.

The RISC-V MC is the microcontroller version. This model balances performance with area. The MC model supports optional features for performance improvement, including the RISC-V M extension (hardware-based integer multiply and divide) and instruction and data caches. This model also optionally supports the RISC-V C extension that uses 16-bit compressed instructions to save code space. The MC model uses AHB-Lite as the native interconnect and is suitable for use in smaller device families such as MachXO2 and MachXO3.

The RISC-V RX is the highest performing model in the Lattice RISC-V family. This model is RTOS capable as this model adds Supervisor and User modes from the Privileged Architecture portion of the RISC-V ISA. The RX model uses AXI4 as the native interconnect for higher performance and more deterministic timing. This model also adds a Platform Level Interrupt Controller (PLIC) and a Core Local Interrupter (CLINT) for managing external interrupts and timers, respectively. A watchdog timer and optional UART are integrated into the RX IP. This IP also supports custom instructions via the Composable Custom Extension, an emerging industry standard.

Table 2.2 shows the features comparison for every RISC-V variant provided by Lattice.

Table 2.2. Features Comparison of Lattice RISC-V Variants

Variant	Mode	Extensions[2]	System Bus	Interrupt Controller	Branch Prediction	System Timer	Cache[3]	Debug Module	Privilege Mode[4]	Custom Instructions	PMP	Soft JTAG
RX	Advanced	I, M, C, F, A	AXI4[6], AHB-L[1] and Local Bus[1]	PLIC[1]	Dynamic Target[6]	CLINT[1]	Yes	Yes[1]	U, S. M	Yes[1] (CXU-LI)	Yes	Yes[1]
	Balanced	I, M, A			Dynamic Target[6]		Yes		U, M		—	
	Lite	I, M, C			—		—		U, S, M		—	
MC	—	I, M[1], C[1], E[1]	AHB-L	PIC [1]	Dynamic Target[6]	Memory Mapped mtime	Yes[1]	Yes[1]	M	Yes[1] (CXU-LI)	—	Yes[1]
SM	—	I	AHB-L	PIC[1]	—	Memory Mapped mtime	—	Yes[1]	M	—	—	Yes[1]
NANO	—	I, C[1]	AHB-L	LWIMC	—	—	—	—	M	—	—	—

Notes:

1. Optional.
2. I: Integer, M: Multiply/Divide, C: Compressed Instruction, F: Single-precision Floating-point, A: Atomic Instruction, E: Embedded.
3. 4096 bytes size, 2-way set associative, 32-byte per cache line, write through policy.
4. U: User Mode, S: Supervisor Mode, M: Machine Mode.
5. Design with Dynamic Target Branch Prediction, requires launching the Lattice Radiant software from the Lattice Propel software to update project setting.
6. Support atomic access on the AXI4 bus.

Refer to the IP user guide listed in the References section for the latest resource utilization (LUT usage) and performance data (including Fmax and DMIPS) for each RISC-V CPU variant.

3. RISC-V Embedded Hardware Design Guidelines

This section describes the options when designing RISC-V embedded hardware systems using the Lattice Propel Builder.

3.1. RISC-V Processor Reset and Exception Vector

RISC-V processor executes the memory address set by reset vector after released from reset. Table 3.1 shows the supported reset vectors for RISC-V NANO, SM, MC, and RX processors. Assign the memory that contains initial software to address matching the reset vector shown in the table.

Table 3.1. RISC-V Processor Reset Vector

RISC-V Processor Variant	Reset Vector[1]
NANO	Fixed (0x0000_0000)
SM	Configurable
MC	Configurable
RX	Configurable for Balanced and Advanced Fixed (0x0000_0000) for Lite

Note:

1. Based on the Lattice Propel software version 2025.2.

Exception vector is the memory address that contains the exception handler code. The Machine Trap-Vector Base-Address (mtvec) register in the RISC-V processor holds the exception vector. The vector is set by the RISC-V software driver during runtime. The driver is generated as part of the Board Support Package (BSP). Refer to the Interrupts and Exceptions section for more information.

3.2. RISC-V Processor Memory Selection

Memory device is used to store RISC-V processor instructions and data of a software program. The types of memory devices are described in Table 3.2.

Table 3.2. Types of Memory Device

Types of Memory Device	Example	Description
Volatile memory	• Cache • Random access memory (RAM)	• Only retains the data while power is supplied to it. Data is lost when the memory power supply is turned off. • Use as temporary storage and has faster access speed.
Non-volatile memory	• Read-only memory (ROM) • Flash memory	• Retains data even when power is turned off. • Use for storing contents permanently and usually has slower access speed compared to volatile memory.

3.2.1. Cache

Cache memory is a high-speed memory that integrates directly into the RISC-V processor. Cache acts as a temporary storage that processor can retrieve data faster. Table 3.3 shows the cache capabilities of the Lattice RISC-V processors.

Table 3.3. RISC-V Processor Caches

RISC-V Processor Variant	Instruction Cache Size	Data Cache Size	Option to Disable Cache	Cache Range[1]
NANO	—	—	—	—
SM	—	—	—	—
MC	4 Kbytes	4 Kbytes	Yes[2]	User configurable Lower limit: 0x00000000 Upper limit: 0xFFFFFFFF

RISC-V Processor Variant	Instruction Cache Size	Data Cache Size	Option to Disable Cache	Cache Range[1]
RX (Advanced/Balanced)	4 Kbytes	4 Kbytes	No	User configurable Lower limit: 0x00000000 (fixed) Upper limit: 0xC0000000

Notes:

1. Based on the Lattice Propel software version 2025.2.
2. Enabling cache may improve Fmax for MC variant.

Enabling the processor caches consumes FPGA memory resources for better CPU performance. It is recommended to enable cache for design with Lattice Avant, MachXO5-NX, CrossLink-NX, CertusPro-NX, and Certus-NX devices.

3.2.2. Tightly-Coupled Memory (TCM)

Tightly-coupled memory provides the processor low-latency predictable access for critical instruction and data. The Lattice Propel software provides TCM IP that can be used with RISC-V RX processor. The TCM IP supports local bus interface which is connected directly to the RISC-V RX local instruction and data ports. This direct connection provides the RISC-V RX low-latency and predictable access to the memory. TCM uses FPGA memory resources like the system memory.

When using TCM, the first 2 Mbytes of processor address range (0x00000000 to 0x001F_FFFF) is reserved for TCM. If system memory (or other types of memory) is used for the processor non-TCM instruction port (for example the AXI port), this memory must start from address 0x0020_0000.

3.2.3. System Memory

System memory provides easy use of Lattice FPGA memory resources (EBR, distributed RAM, or large RAM) as an IP available in the Lattice Propel software. System memory does not require connections to external devices from FPGA and can store RISC-V software and data using FPGA resources.

Table 3.4 shows the system memory IP features and use cases.

Table 3.4. System Memory IP Features and Use Cases

Feature	Use Case
AHB-Lite interface	Use as memory for RISC-V NANO, SM, and MC
AXI4 interface	Use as memory for RISC-V RX
Configurable as single or dual port memory	With dual port memory, the RISC-V instruction port can be connected directly to one of the memory ports. The other memory port is for RISC-V data port.
Memory initialization enable	Memory is initialized with RISC-V software during FPGA configuration process. Processor executes instructions stored in memory when out of reset. Typically, use the bootloader software in this scenario.

3.2.4. External SDRAM

External SDRAM can be used in applications that require larger memory capacity. Large programs and data sets can be stored and accessed on the external memory, though with increased latency. SDRAM requires a controller to manage refresh operations and handle memory accesses, such as switching between banks, rows, and columns.

The Lattice Propel software offers SDRAM memory controllers for CertusPro-NX and Avant devices. Both memory controllers support LPDDR4 SDRAM and include AXI4 interface for processor connectivity.

Table 3.5 shows the memory controller parameters that you can customize for RISC-V applications.

Table 3.5. LPDDR4 SDRAM Memory Controller Parameters

Parameter	Description
DDR Command Frequency (MHz)	Change the frequency to the desired value that matches your design. The maximum frequency is 800 MHz for Lattice Avant devices and 533 MHz for CertusPro-NX devices.
DDR Density	Change the density (in terms of Gb) per channel to match the SDRAM chip. This change

Parameter	Description
	affects the address bus width of the controller AXI4 interface.
DDR Bus Width	Change the bus width for DDR data bus to match the SDRAM chip. This change affects the DQ, DQS, and DMI bus width, and requires update to FPGA pin assignments.
Data Width on Local Data Bus	Change the data bus width (range from 32-bit to 256-bit) of the AXI4 interface. RISC-V processor has 32-bit data bus. To avoid width adaptation, set this parameter to match the processor width.

3.2.5. Flash Memory

Flash memory is a non-volatile memory used to store permanent data. Most embedded applications rely on flash memory for storing processor programs (such as bootloaders) and data (such as media files and configuration files). Unlike volatile memory (such as SDRAM), writing to flash requires erasing the corresponding page before performing a write operation. To enable RISC-V processors to access flash memory, a flash memory controller is required.

The Lattice Propel software provides the SPI Flash Controller IP that connects the RISC-V processor to a SPI-based flash device. This IP supports a single-bit serial data interface and offers an AHB-Lite interface for data access and an APB interface for control register access. When connecting the RISC-V RX processor to the AHB-Lite port, a converter bridge is needed.

For correct data capture from the SPI flash, the input delay of the MISO signal must be properly constrained. The set_input_delay constraint defines the delay introduced by the flash device and PCB traces based on the flash datasheet and board layout. The example constraints are as follows:

- `set_input_delay -clock [get_clocks spi_flash0_inst_spi_clk] -max 8 [get_ports miso_i]`
- `set_input_delay -clock [get_clocks spi_flash0_inst_spi_clk] -min 6 [get_ports miso_i]`

The values from the example constraints (8 ns and 6 ns) are not arbitrary. They represent the latest and earliest arrival times of MISO data relative to the SPI clock edge. They are calculated as follows:

- InputDelay_max = SCLK forward delay + Flash tCO_max + MISO return delay
- InputDelay_min = SCLK forward delay + Flash tCO_min + MISO return delay

For example:

- Flash clock-to-output delay (tCO): max = 6.0 ns, min = 5.0 ns (from datasheet)
- PCB trace delays: SCLK forward ≈ 0.8 ns, MISO return ≈ 1.0 ns

The input delays are calculated as follows:

- InputDelay_max = 0.8 + 6.0 + 1.0 ≈ 7.8 ns (rounded up to 8 ns)
- InputDelay_min = 0.8 + 5.0 + 1.0 ≈ 6.8 ns (rounded down to 6 ns)

Clarification on rounding:

To ensure conservative timing analysis, the maximum delay must be rounded up (round up 7.8 ns to 8 ns) to cover the latest possible arrival, while the minimum delay must be rounded down to the next lower integer (round down 6.8 ns to 6 ns) to cover the earliest possible arrival. This rounding is based on absolute time values in nanoseconds, not on SPI frequency or clock period.

The reason to round minimum delay down instead of up:

The minimum value represents the earliest possible arrival of data. If you round it up, you make the data appears later than the data can arrive, which may hide a real hold violation. By rounding down, you assume the data can arrive even earlier, which is conservative for hold timing. In contrast, the maximum value is for setup checks, so rounding up makes the data appears later, which is conservative for setup timing.

3.3. Interconnects and Bridges

Interconnects connect multiple managers to multiple subordinates within a system. Examples of managers include RISC-V instruction port, RISC-V data ports, and DMA. Subordinates typically include peripherals such as UART, SPI, and I2C controllers. Interconnect provides several key functions, including:

- Decode the managers transaction address and routing it to the correct subordinate.
- Arbitrate concurrent transactions from multiple managers targeting a specific subordinate.
- Convert different data widths between managers and subordinates.

The Lattice Propel software provides interconnect solutions compliant with the Arm Advanced Microcontroller Bus Architecture (AMBA), including AXI4, AXI4-Lite, AHB-Lite, and APB protocols. Select the appropriate interconnect IP for your project. The following sub-sections describe each interconnect solution in detail.

Bridges are required to connect different AMBA interfaces connect interfaces that use different AMBA protocols. For example, connecting an AXI4 manager to an APB subordinate. Bridge IP simplifies system building by allowing managers and subordinates to communicate without modifying the AMBA interface.

3.3.1. Unified Interconnect

The Unified Interconnect IP is introduced as a next-generation solution designed to replace the existing connectivity medium across multiple AMBA protocols. This IP enables efficient and seamless communication between managers and subordinates in memory-mapped designs, addressing the growing complexity of modern SoC architectures.

In the Lattice Propel Builder 2025.2, the initial release of Unified Interconnect IP delivers functionality equivalent to the AXI4 Interconnect IP, while introducing key improvements such as enhanced transaction routing latency and optimized logic utilization. These enhancements aim to improve overall system performance and resource efficiency.

The Unified Interconnect IP v1.0.0 supports only a unified data width configuration and is restricted to AXI4 and AXI4-Lite protocol interfaces. These constraints are intended to simplify integration and validation for early adopters while laying the foundation for broader protocol support in future versions.

For comprehensive details on supported features, configuration options, and usage guidelines, refer to the Unified Interconnect IP User Guide (FPGA-IPUG-02318). The guide also outlines the planned migration from AXI4 Interconnect IP to Unified Interconnect IP if the released features meet the system design requirements.

3.3.2. Using AXI4 Interconnect IP

The AXI4 Interconnect IP connects multiple AXI4 based managers to AXI4 subordinates. This interconnect is useful for the Lattice RISC-V RX processor-based system as both the instruction and data ports are AXI4 interfaces. The AXI4 Interconnect IP supports both AXI4 and AXI4-Lite protocols. The protocol type can be configured for individual manager and subordinate.

The AXI4 Interconnect IP is fully parameterizable for your application. Figure 3.1, Figure 3.2, and Figure 3.3 show the IP Wizard for the configurable parameters. The subsequent subsections discuss the guidelines when configuring the AXI4 Interconnect IP.

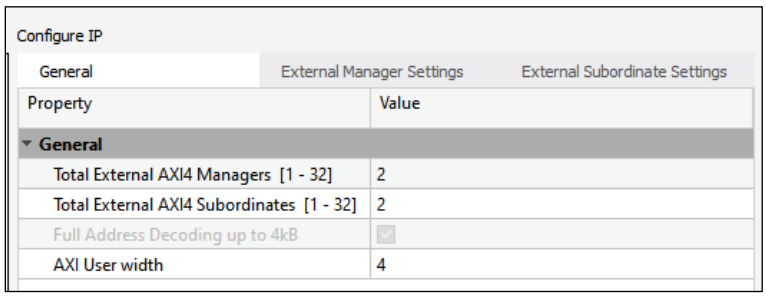

Figure 3.1. AXI4 Interconnect IP Parameters – General Tab

Configure IP

General | External Manager Settings | External Subordinate Settings

Property	Value
General	
External Manager AXI ID width	1
AXI Manager Max Address Width(bits)	32
AXI Manager Max Data Width(bits)	32
AXI Manager Max no.of ID supports	16
External Manager Access Type Settings	
INFO: Ext Manager Access type list	{2'd2,2'd2}
External Manager AXI Access Type 0	WR
External Manager AXI Access Type 1	WR
External Manager Protocol Settings	
INFO: Ext Manager Protocol type list	{1'd0,1'd0}
External Manager AXI protocol 0	AXI4
External Manager AXI protocol 1	AXI4
External Manager CDC Enable Settings	
INFO: Ext Manager CDC Enable list	{1'd0,1'd0}
External Manager CDC Enable 0	☐
External Manager CDC Enable 1	☐
External Manager Address Settings	
INFO: Ext Manager actual address list	{7'd32,7'd32}
External Manager Address width 0	32
External Manager Address width 1	32
External Manager Data Settings	
INFO: Ext Manager actual Data width list	{11'd32,11'd32}
External Manager Data width 0	32
External Manager Data width 1	32
External Manager No.of IDs supports Settings	
INFO: Ext Manager ID supports list	{7'd16,7'd16}
External Manager No.of IDs 0	16
External Manager No.of IDs 1	16

Figure 3.2. AXI4 Interconnect IP Parameters – External Manager Settings Tab

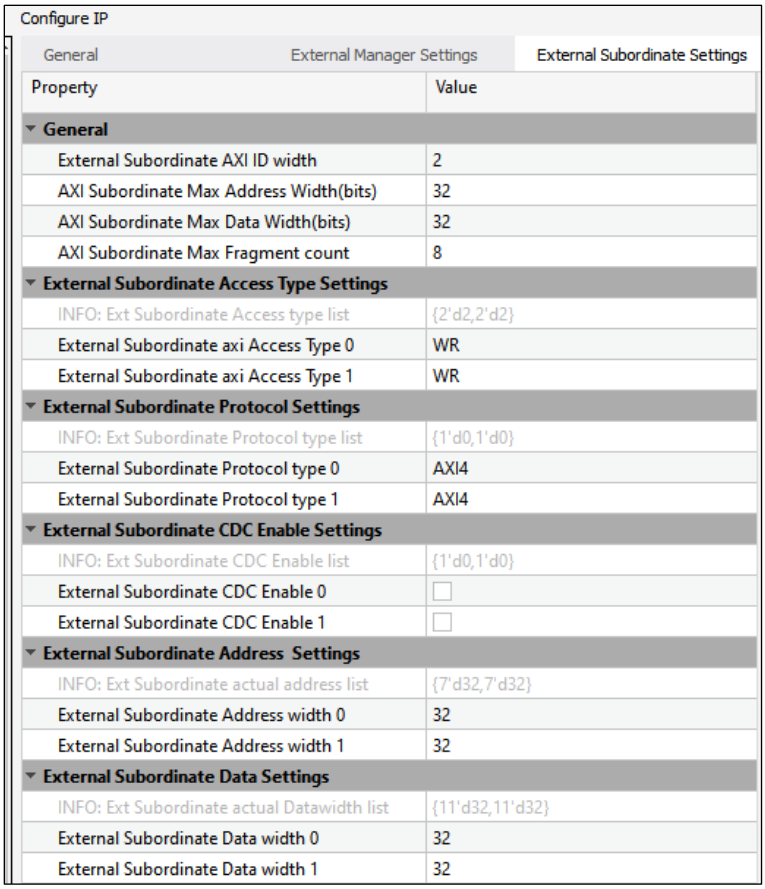

Figure 3.3. AXI4 Interconnect IP Parameters – External Subordinate Settings Tab

3.3.2.1. Total External Managers and Subordinates

The AXI4 Interconnect IP is configured based on the total number of external managers and subordinates in the system.

The **Total External AXI4 Managers** parameter is set to match with total managers in the system. For example, RISC-V RX has 2 managers (instruction and data). If other managers (such as DMA) are connected to the AXI4 Interconnect IP, increase this number accordingly.

The **Total External AXI4 Subordinates** parameter is set to match with total subordinates in the system that needs to connect with the managers.

Total managers and total subordinates cannot have the values of 1 because one manager and one subordinate can be connected directly without an interconnect.

3.3.2.2. AXI Address and Data

The AXI4 Interconnect IP supports configurable address and data widths to match with all external managers and the connected subordinates. Set the maximum allowable address and data width on both the manager and subordinate using the following parameters:

- **AXI Manager Max Address Width**
- **AXI Manager Max Data Width**
- **AXI Subordinate Max Address Width**
- **AXI Subordinate Max Data width**

For each external manager interface, set the actual **External Manager Address Width** and **Data Width** parameters to match with the manager properties. Similar approach applies when setting the **External Subordinate Address Width** and **Data**

Width parameters. For example, the RISC-V RX processor address width is 32 bit and data width is 32 bit. The width value must not exceed the values set in the maximum allowable parameters.

When the manager and subordinate data widths are not equal, the AXI4 Interconnect IP inserts width converter to handle the differences. For example, when connecting the RISC-V RX processor (data width is 32 bit) to LPDDR4 memory controller subordinate (data width is 256 bit), the interconnect applies width conversion. Note that width converter increases the logic utilization and may impact the timing performance of the achievable Fmax. For optimizing the interconnect usage, configure the managers and subordinates to the same width when possible.

3.3.2.3. AXI USER

AXI User signal is for user-defined purposes. The AXI4 Interconnect IP supports configurable **AXI User Width** parameter, which applies to all manager and subordinate interfaces within the IP. Set the AXI User signal (AxUSER, xUSER, and BUSER) width to match the application.

For AXI4 manager to AXI4-Lite subordinate, the AXI4 Interconnect IP ignores the AXI User signals when routing the transactions.

For AXI4 manager to AXI4 subordinate, the AXI4 Interconnect IP passes through the AXI User signals without changing the signals.

3.3.2.4. AXI ID

The AXI ID signals are used for transaction identification and ordering. The AXI4 Interconnect IP supports configurable AXI ID width for external managers and subordinates. The **External Manager AXI ID Width** and **External Subordinate AXI ID Width** parameters set the ID widths to match the following equation:

$$\text{Subordinate ID width} \geq \text{External Manager AXI ID Width} + \max(\log2(\text{Total external AXI4 managers}), 1)$$

For example, on system with RISC-V RX processor where RISC-V RX manager ID width = 1 and number of managers = 1, the subordinate ID width is set to 2 using the equation as follows:

$$\begin{aligned}\text{Subordinate ID width} &\geq \text{External Manager AXI ID Width} + \max(\log2(\text{Total external AXI4 managers}), 1)\\ &\geq 1 + \max(\log2(1), 1)\\ &\geq 2\end{aligned}$$

The AXI Interconnect IP external subordinate ID width must match with the actual subordinate. For example, if the Interconnect IP ID width is 2, the subordinate on system memory IP must match. If the subordinate ID width does not match, the extra bits on ID signal are left undriven and may lead to unpredictable behavior.

The **AXI Manager Max no of ID Supports** parameter sets maximum allowable number of ID for external managers arriving at the interconnect. The **External Manager No of ID** parameter is set for each manager interface to change the interconnect reordering depth for transactions ID tracking. The value set for this parameter must not be larger than the maximum allowable number of ID parameter.

3.3.2.5. Clock Domain Crossing

The AXI4 Interconnect IP operates at single clock domain which is driven by the **axi_aclk_i** clock input signal. For system with single clock domain, the axi_aclk_i is connected to the same clock as with the rest of the system.

However, some external managers or subordinates operate at different clock than the axi_aclk_i. The AXI4 Interconnect IP supports clock domain crossing that can be enabled on individual manager or subordinate interface. The **External Manager CDC Enable** and **External Subordinate CDC Enable** parameters are enabled according to the system clock crossing setup.

Figure 3.4 shows RISC-V RX, AXI4 interconnect, and memory operate in CLK A domain while DMA operates in CLK B domain. DMA is connected to the AXI interconnect where the DMA accesses to the memory that operates in CLK A domain.

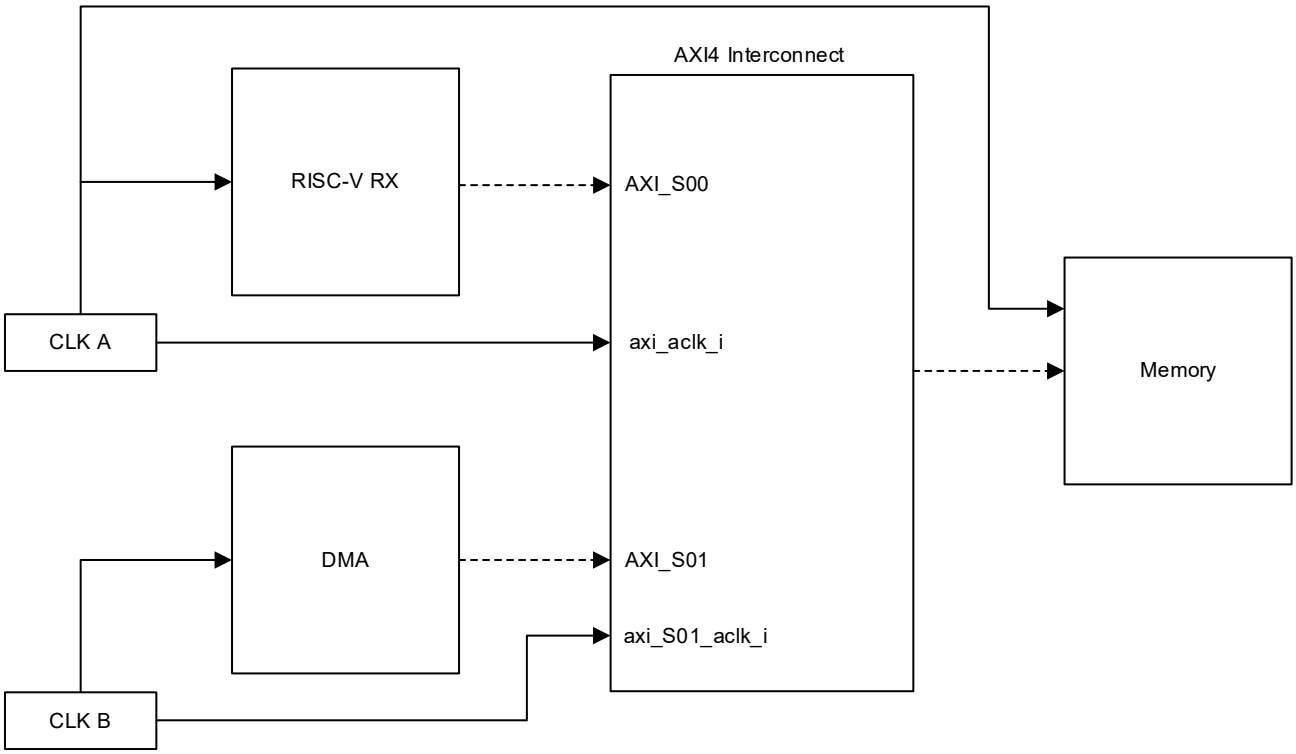

Figure 3.4. Example System with Different Clock Domains

DMA is connected to AXI_S01 port of the AXI4 interconnect. To turn on clock domain crossing for this port, enable the **External Manager CDC Enable 1** parameter. Additional clock and reset signals listed in Table 3.6 are exported when the CDC feature is enabled.

Table 3.6. AXI4 Interconnect CDC Signals

CDC Signal	Description
axi_S01_aclk_i	Clock input for AXI_S01 port. Connect this signal to the clock that external manager uses (CLK B).
axi_S01_aclken_i	Clock enable input (active low) for AXI_S01 port. Connect to the clock source enable signal. For example, if the clock is from PLL, connect the PLL Lock output signal to this signal.
axi_S01_aresetn_i	Reset input (active low) for AXI_S01 port. This reset signal is de-asserted synchronously to axi_S01_aclk_i.

3.3.3. Using AHB-Lite Interconnect IP

The AHB-Lite Interconnect IP connects multiple AHB-Lite based managers to AHB-Lite subordinates. This interconnect is useful for Lattice RISC-V MC and RISC-V SM processor-based systems as both the instruction and data ports are AHB-Lite interfaces.

The AHB-Lite Interconnect IP is fully parameterizable for your applications. Figure 3.5 shows the IP Wizard with the configurable parameters. The subsequent subsections discuss the guideline when configuring the AXI4-Lite Interconnect IP.

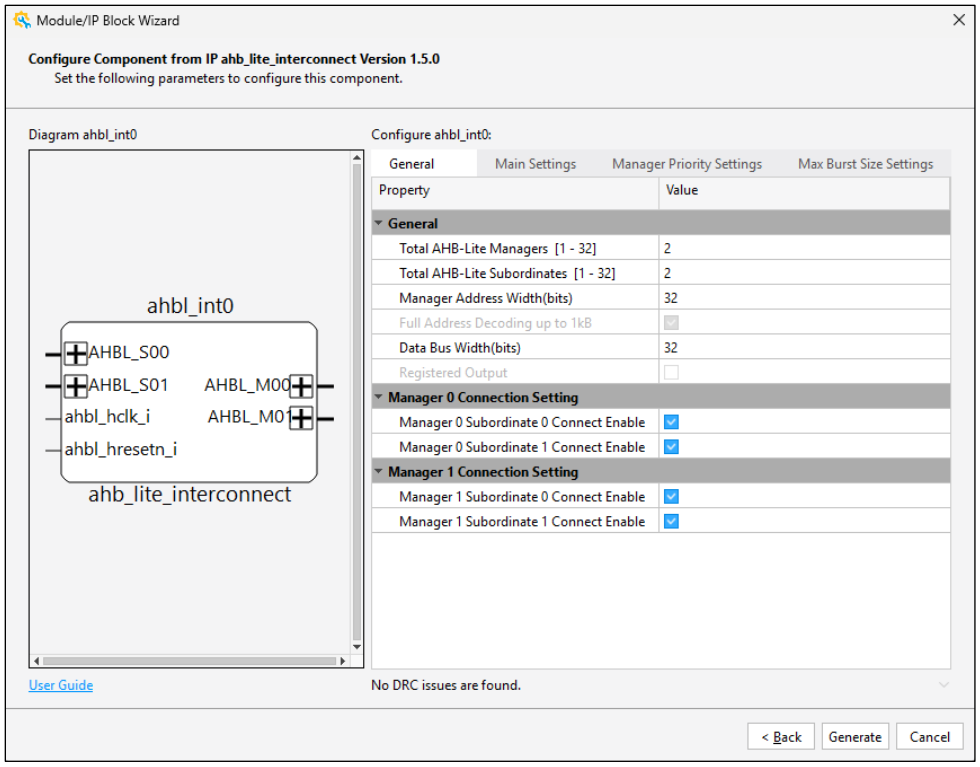

Figure 3.5. AHB-Lite Interconnect IP Parameters – General Tab

3.3.3.1. General Settings

The **Total AHB-Lite Managers** and **Total AHB-Lite Subordinates** parameters set the numbers of managers and subordinates that are connected to the interconnect. The RISC-V MC, SM, or NANO processor has 2 AHB-Lite manager ports (Instruction and Data). Only the Data port requires connection to multiple subordinates (peripherals). The Instruction port is usually connected directly to memory that stores the program. When the Instruction port requires access to a shared memory, the AHB-Lite Interconnect IP increases the value for **Total AHB-Lite Managers** and connects the Instruction port to the AHB-Lite interconnect that grants access to the shared memory.

When the AHB-Lite Interconnect IP is configured with multiple managers and subordinates, all connections between managers and subordinates are enabled by default. In the example showed in Figure 3.5, 2 managers and 2 subordinates result in 4 interconnection logics. If one of the managers does not require access to a specific subordinate, the **Connect Enable** checkbox can be disabled. For example, if Manager 0 does not access Subordinate 1, uncheck **Manager 0 Subordinate 1 Connect Enable** to reduce the logic generated for the interconnect and improve overall Fmax.

3.3.3.2. Main Settings

The **Main Settings** tab lists the default base address and range for each subordinate that is enabled on the AHB-Lite Interconnect IP. The actual addresses are not set via parameters in this setting tab.

When creating RISC-V system with the Lattice Propel Builder, each subordinate base address and range are set up in the **Address** tab of the Lattice Propel Builder GUI. This address information is then propagated to the AHB-Lite Interconnect IP during system generation in the Lattice Propel Builder. The generated system (RTL code) parameterizes the AHB-Lite Interconnect IP with the addresses from the Lattice Propel GUI during IP instantiation.

3.3.3.3. Manager Priority Settings

The **Manager Priority Settings** tab allows selection of the AHB-Lite subordinates arbitration scheme. Select **either Round Robin** or **Fixed Priority** scheme to match the application requirements.

3.3.3.4. Max Burst Size Settings

The **Max Burst Size Settings** tab allows selection of the maximum burst size for each AHB-Lite subordinates in the interconnect. If the manager performs burst transaction on one subordinate and the transaction exceeds the said maximum burst size setting, the interconnect sends an error response to the manager. This prevents a manager from hogging the bus by generating very long bursts.

3.3.4. Using APB Interconnect IP

The APB interface is commonly used for low bandwidth bus interface such as peripherals access. In a typical embedded system, the processor interfaces to multiple peripherals such as I2C controller, SPI flash controller, LPDDR4 memory controller, UART, and GPIO. These peripherals do not require high performance bus and typically use the APB interface.

RISC-V RX has AXI4 interface and RISC-V SM, MC, or NANO has AHB-Lite interface. The processors use conversion bridges such as AXI4-to-APB bridge to connect the processor interface to the APB-based peripherals. These bridges operate on a per interface basis where a bridge is required for each peripheral. APB Interconnect IP allows a single manager to connect to multiple subordinates, eliminating the need for individual bridge.

Figure 3.6 shows the RISC-V RX data port accessing GPIO, SPI flash controller, and LPDDR4 memory controller. All these peripherals have APB based subordinate interface. A single AXI-to-APB bridge converts the AXI interface from the AXI4 Interconnect to APB interface. Subsequently, the APB Interconnect connects the interface to multiple peripherals.

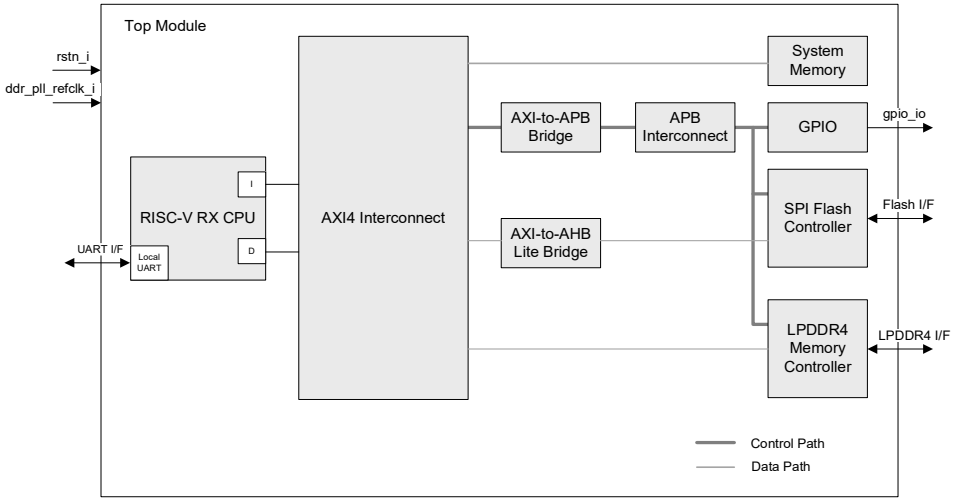

Figure 3.6. System with APB Interconnect for Peripherals Access

3.3.5. Using Bridges IP

Bridges connect different types of AMBA interfaces in the Lattice Propel system. The following bridges are available in the Lattice Propel Builder:

- AXI4-to-AHB-Lite bridge
- AXI4-to-APB bridge
- AHB-Lite-to-APB bridge
- AHB-Lite-to-AXI4 bridge

3.3.5.1. AXI4-to-AHB-Lite Bridge

This bridge provides a connection between the AXI4 interface and any AHB-Lite subordinate interface.

The **AXI_AHB_DATA_WIDTH** parameter is configurable from 8 bits up to 1024 bits, with a default value of 32 bits. When the RISC-V processor uses a 32-bit data width, this parameter must be set to 32.

The **AXI_ID_WIDTH** parameter can be set between 1 and 11 bits and must match the width used by the upstream manager. For example, if the upstream AXI4 interconnect uses an ID width of 2, the bridge must also be configured to 2.

Similarly, the **AXI_USER_WIDTH** parameter, which can range from 1 to 128 bits, this parameter must be set to match the upstream manager.

The AXI4-to-AHB-Lite bridge does not support clock domain crossing internally. If the AXI4 and AHB-Lite interfaces operate in different clock domains, this bridge must be handled externally. When an AXI4 interconnect is used upstream, clock domain crossing can be enabled in that interconnect. For more details, refer to the Using AXI4 Interconnect IP section.

3.3.5.2. AXI4-to-APB Bridge

This bridge connects the AXI4 interface to any APB subordinate interface.

The **AXI_APB_DATA_WIDTH** parameter supports values of 8, 16, or 32 bits, with 32 bits as the default. For a RISC-V processor with a 32-bit data width, this parameter must be set to 32.

The **AXI_ID_WIDTH** parameter is configurable from 1 to 11 bits and must match the upstream manager.

The **AXI_USER_WIDTH** parameter, which ranges from 1 to 128 bits, must also align with the upstream manager.

The AXI4-to-APB bridge does not provide internal clock domain crossing support. If the AXI4 and APB interfaces operate in different clock domains, this bridge must be managed externally. When an AXI4 interconnect is used upstream, clock domain crossing can be enabled in that interconnect. For more details, refer to the Using AXI4 Interconnect IP section.

3.3.5.3. AHB-Lite-to-APB Bridge

This bridge connects an AHB-Lite interface to any APB subordinate interface.

When interfacing with RISC-V processors, the **ADDR_WIDTH** and **DATA_WIDTH** parameters must be configured to 32 bits.

The bridge supports an optional separate APB clock domain, which allows the APB interface to operate independently from the AHB-Lite clock domain. This feature is useful for decoupling slower peripherals from the system clock domain and can help achieve timing closure more easily.

To enable this functionality, the **APB Clock Enable** parameter must be checked. This feature adds separate clock and reset signals for the APB domain (pclk_i and presetn_i) and generates the necessary clock domain crossing logic. For more details, refer to the *Attributes* section of the AHB-Lite to APB Bridge Module User Guide (FPGA-IPUG-02053).

3.3.5.4. AHB-Lite-to-AXI4 Bridge

This bridge provides an interface between a single AHB-Lite manager and an AXI4 subordinate. This bridge translates read and write transfers on the AHB-Lite side into equivalent AXI4 transactions. Some additional wait states are expected on the AHB-Lite bus because of translation and pipeline operations.

The **M_ADDR_WIDTH** parameter specifies the address bus width for both AHB-Lite and AXI4 interfaces. This parameter is configurable from 11 to 32 bits, with a default value of 32 bits.

The **DATA_WIDTH** parameter defines the data bus width for both interfaces. This parameter supports values of 8, 16, 32, 64, 128, 256, 512, and 1024 bits, with a default of 32 bits. When the RISC-V processor uses a 32-bit data width, this parameter must be set to 32.

The **AXI_ID_WIDTH** parameter is configurable from 1 to 11 bits and determines the width of AXI4 transaction ID signals. The ID value for this bridge is always tied to 0 because AHB-Lite does not support transaction IDs.

The **AXI_SECURE_ACCESS** parameter enables secure access for all AXI4 transactions when set to 1. By default, this parameter is disabled (0).

The **AXI_TIMEOUT** parameter specifies the number of clock cycles to wait for an AXI4 subordinate response before asserting a timeout signal. A value of 0 disables the timeout feature, while selectable values include 16, 32, 64, and 128 cycles.

The **RDDATA_PIPELINE** and **WRDATA_PIPELINE** parameters allow adding pipeline stages to the read and write data buses respectively. Enabling these parameters can help relax timing on critical paths. Both default to 0 (disabled).

The **NARROW_TRANSFER** parameter enables support for narrow transfers where the requested transfer size is smaller than the configured **DATA_WIDTH**. This option is available only when **DATA_WIDTH** is greater than 8 bits.

The AHB-Lite-to-AXI4 bridge does not support clock domain crossing internally. Both interfaces are expected to operate in the same clock domain and share the same address and data bus widths. For more details, refer to the *Attributes* section of the AHB-Lite to AXI4 Bridge IP User Guide (FPGA-IPUG-02242).

3.4. Integrating External IP via Feedthrough IP

The primary purpose of the feedthrough IP is to enable address mapping in the Lattice Propel Builder for signals that need to be exported and connected to a higher-level module, such as the module in Lattice Radiant project. This is useful when integrating IP that is not available within the Lattice Propel Builder. For example, you can route the exported bus to an IP that is available in the Lattice Radiant software. This enables the integration of external IP by ensuring proper address mapping for the exported signals. The following feedthrough IPs are available in the Lattice Propel Builder:

- AHB-Lite Feedthrough IP
- APB Feedthrough IP
- AXI4 Feedthrough IP

Figure 3.7 shows the Lattice Propel Builder system with the feedthrough IP. The AHB-Lite feedthrough (ahbl_feed_inst) is connected from the AHB-Lite interconnect for the processor to access to external IP. Similarly, the APB feedthrough (apb_feed_inst) is connected to APB interconnect located at the top right of Figure 3.7.

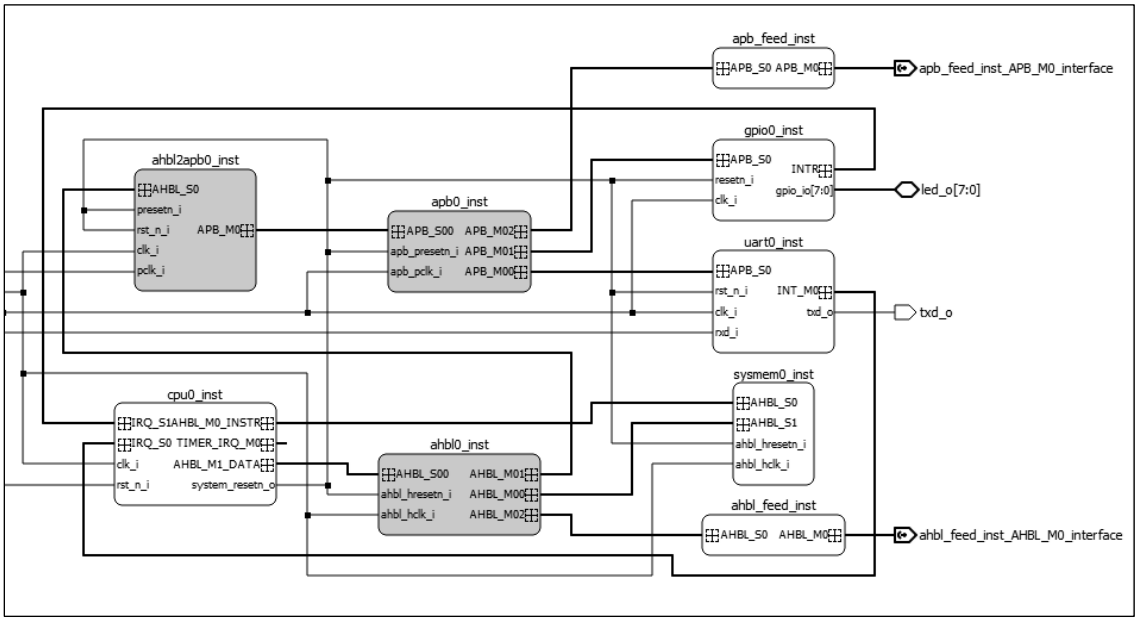

Figure 3.7. Lattice Propel Builder System with Feedthrough IP

When connecting to the external subordinate, set **Export Interface As** to **Subordinate** in the Bridge Module/IP Block Wizard as shown in Figure 3.8.

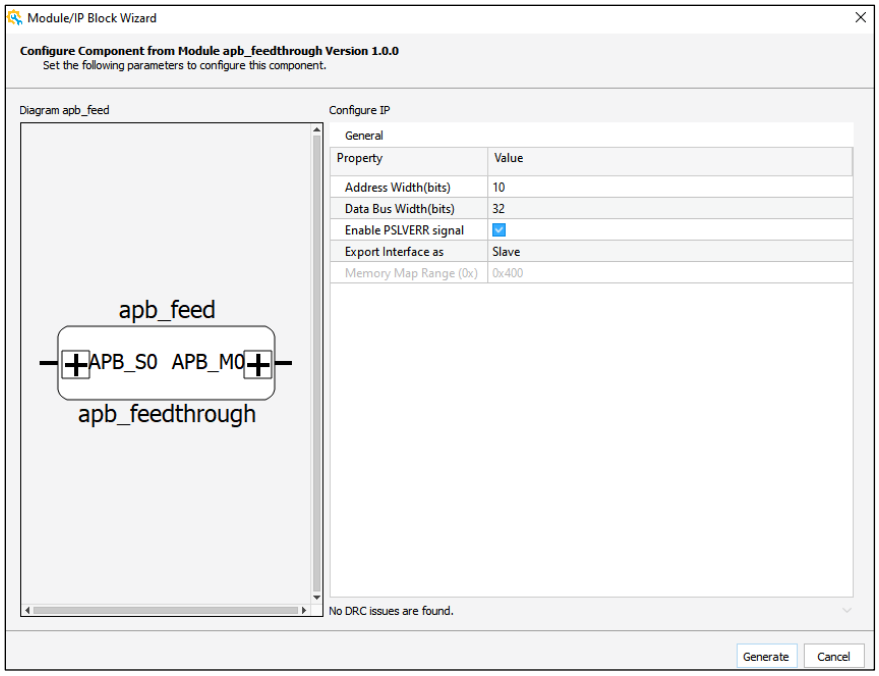

Figure 3.8. APB Feedthrough Module/IP Block Wizard

To create a memory-mapped address space for the feedthrough block, click **Generate** in the bridge Module/IP Block Wizard in Figure 3.8. When interface to RISC-V processor, the software code accesses the block via the assigned address. Figure 3.9 shows the address space for the APB feedthrough in the **Address** tab in the Lattice Propel Builder.

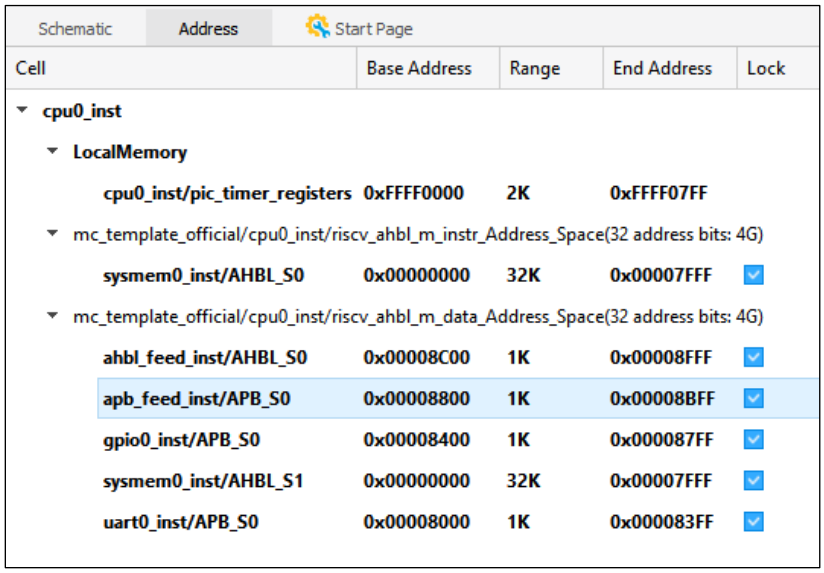

Figure 3.9. Address Map of Feedthrough IP

3.5. Mailbox IP

The Mailbox IP enables bi-directional, FIFO-based messaging between two processors (or subsystems) in a multi-processor SoC. This IP provides ordered write and read semantics, status and error reporting, and interrupt-driven flow control via configurable thresholds.

Use mailbox for low-overhead control-plane communication between cores, especially for task dispatch and result collection or event signaling, where ordered delivery and simple backpressure are desirable without managing shared-memory protocols.

The design recommendations are as follows:

- Use mailbox for control-plane traffic. Set the mailbox for small, fixed-size messages or tokens between cores; keep bulk data in shared memory where appropriate.
- Treat mailbox as MMIO. Map the registers into a non-cacheable region and access via normal loads or stores to avoid stale status or data reads.
- Combine with concurrency and protection blocks. Pair mailbox with a mutex when multiple cores share peripherals (for example, UART) and apply your system interrupt and protection policies (for example, PIC, IOPMP) for clean routing and isolation.
- Start with polling, then move to IRQs. Use polling during bring-up; migrate to interrupt-driven handlers when flows stabilize. Set conservative thresholds initially (to prevent underflow or overflow), tune the thresholds after you observe how quickly two endpoints (for example, CPU0 and CPU1) send and receive messages.

For more information, refer to the Mailbox IP User Guide (FPGA-IPUG-02306). An example design demonstrates how mailbox is implemented in a multi-processor SoC and can be generated from the RISC-V MC Multi-Processor Project template.

3.6. Mutex IP

The Mutex IP provides mutual exclusion in multi-processor SoCs via configurable mutex registers. Each register contains a lock bit plus a CPU identification field, and an optional port protection feature that ties unlock permission to the original locking port. Interfaces per instance can be AXI-Lite or AHB-Lite.

Use mutex whenever multiple CPUs share a resource that must be accessed by one CPU at a time. Typical examples include a shared UART or a common MMIO register window.

The design recommendations are as follows:

- Use mutexes at clear resource boundaries (for example, UART transmit or shared register window) and keep critical sections short to minimize blocking.
- Follow a lock-then-verify pattern: Write your ID while setting **Lock = 1**, read back to confirm ownership before using the resource. On release, write ID with **Lock = 0** and read back to confirm the IP is free.
- Port protection: Enable port protection when many CPUs use the mutex to ensure only the CPU that locked it can unlock it, even if another CPU tries to write the same ID.
- CPU ID width: Set the CPU ID width large enough to give every CPU a unique ID, with additional width for future CPUs.
- Treat mutex registers as MMIO and map them in a non-cacheable region to ensure fresh lock state on reads.
- Pair a mutex with a mailbox when you need both exclusive access to a shared peripheral and message passing for coordination.

For more information, refer to the Mutex IP User Guide (FPGA-IPUG-02307). An example design demonstrates how mutex handles shared resources in a multi-processor SoC and can be generated from the RISC-V MC Multi-Processor Project template.

3.7. IOPMP IP

The RISC-V IOPMP IP blocks illegal or unexpected access to protected memory regions made by non-CPU bus managers (for example, DMA or Ethernet). Both AXI4 and AHB-Lite variants share the same core behavior as follows:

- **Compact-K model**: One memory domain with four programmable entries.
- **Top of Range (TOR)**: Address mode with 4-byte granularity.
- **Per-entry permissions**: Read and write.
- **Violation handling**: Error response or predefined success, plus optional interrupt.
- **Policy hardening**: Entry locking and a sticky register-level lock.

Table 3.7. Comparison Between AXI4 IOPMP and AHB-Lite IOPMP

Feature	AXI4 IOPMP	AHB-Lite IOPMP
Default enable behaviour	Protection check is enabled by default. Traffic from the AXI manager is initially routed to an error subordinate until registers open the normal data path.	Feature is disabled by default. Firmware must set HWCFG0.enable before protection checks become active.
Identity tracking for violations	Uses the AXI ID from the manager. The IP maps the ID to a RRID and logs it when access is denied.	AHB-Lite carries no ID. A fixed RRID scheme is used and an ERROR ID attribute tags denial records per IOPMP instance.
Interface partitioning	One AXI-Lite control interface (registers) plus two AXI data interfaces (checked path), defaults to error routing until entries permit access.	Three AHB-Lite interfaces: one control subordinate (registers) and two data-path interfaces. Protection applies only after enable.
Control-port specifics	AXI-Lite control follows standard AXI-Lite responses (OKAY, SLVERR, DECERR).	Control subordinate accepts single transfer with HTRANS=2'b10 and reports HRESP=0 (OK).
Configuration attributes	AXI ID Width is used for RRID mapping.	ERROR ID is used to tag and distinguish denial records, especially with multiple IOPMP instances.
Programming flow highlights	Set ENTRY_ADDR(i) upper limits and ENTRY_CFG(i) permissions and TOR mode, configure ERR_CFG for violation response and interrupts, lock entries via ENTRYLCK.f and optionally lock ENTRYLOCK. Controller is ON by default.	Same entry setup, permissions, ERR_CFG, and locking, plus an explicit step to enable the module via HWCFG0.enable before checks apply.
TOR range behaviour	TOR defines regions in ascending order. Each entry covers from the previous upper limit up to (but not including) the current upper limit. Out-of-order limits disable the entry.	Same TOR semantics and ordering requirements.
Priority and instruction-fetch checks	Priority not supported. Instruction-fetch checks not supported in this release.	Same limitations in this release.

For more information, refer to the RISC-V AXI4 I/O Physical Memory Protection IP User Guide (FPGA-IPUG-02283) and the RISC-V AHB-L IOPMP IP User Guide (FPGA-IPUG-02286). An example design demonstrates how IOPMP, Mailbox, and Mutex are implemented in a multi-processor SoC and can be generated from the RISC-V MC Multi-Processor Project template. Refer to the Mailbox IP User Guide (FPGA-IPUG-02306) and Mutex IP User Guide (FPGA-IPUG-02307) for the generation flow.

3.8. Assigning Address Map

The Lattice Propel Builder supports assigning addresses to all memory-mapped peripherals in the embedded system. The address map is managed via the **Address** tab in the Lattice Propel Builder. The Auto Assign feature in the Lattice Propel Builder allows the tool to automatically assign non-overlapping addresses to all peripherals. Additionally, when cache is enabled in the RISC-V processor, peripheral address is assigned outside of the cache region automatically, to ensure the processor reads the peripheral registers value from the actual register value and not from the cache. DRC error message is prompted if the peripheral address is configured within the cache region.

3.9. Cache Coherency

In Lattice RISC-V processor-based systems, maintaining cache coherency is essential when the CPU and peripherals such as DMA engines share memory. The processor implements a write-through cache policy, where every store operation updates both the cache and main memory at the same time, while load operations read from the cache if the data is present. When the CPU needs the most recent value from memory, especially after another host has modified the data, the cache line must be invalidated before performing a load.

SGDMA drivers do not manage cache operations and always access memory directly, hence they are unaffected by the processor cacheable address range. However, if SGDMA descriptors or buffers reside in cacheable memory, the CPU must invalidate the cache before reading back status or data to ensure consistency.

Lattice provides the following cache management APIs for software control:

- cache_flush_ins() synchronizes the instruction cache after code changes.
- cache_invalidate(addr) invalidates a specific data cache line.
- cache_invalidate_all() clears the entire data cache.

The following common use cases illustrate best practices:

- For cacheable addresses, CPU writes do not require a flush because of the write-through policy, but invalidation is necessary before reading data updated by another host.
- For non-cacheable addresses, no flush or invalidation is needed. This approach is preferable for DMA buffers to avoid manual cache maintenance.

In summary, cacheable regions improve performance but require explicit cache operations for coherency when shared with peripherals. Non-cacheable regions simplify design by eliminating this requirement. You must carefully assign memory regions and use cache APIs when necessary to maintain data integrity across CPU and DMA interactions.

3.10. Burst Transaction Support

Burst transactions improve memory throughput by transferring multiple beats in a single bus operation. This section summarizes burst transaction capabilities for RX, MC, SM, and NANO cores to guide you in selecting the right CPU for performance-critical applications.

Table 3.8. Lattice RISC-V Processor Burst Support and Cache Details

Feature	RX	MC	SM	NANO
Read burst support	Yes. Supports linear bursts on the local bus when cache is enabled. Each burst is fixed-length with 8 beats (one cache line).	Yes. Supports linear bursts on the AHB-Lite when cache is enabled. Each burst is fixed-length with 8 beats (one cache line).	Not supported.	Not supported.
Other Burst Types (INCR4, INCR16)	Not supported.	Not supported.	Not supported.	Not supported.
Wrap Burst Support	Not supported.	Not supported.	Not supported.	Not supported.
Write Burst Support	Not supported. Write-through single-beat writes to cache and external memory at the same time.	Not supported; write-through single-beat writes to cache and external memory at the same time.	Not supported.	Not supported.
AXI Burst Type	Linear burst with fixed-length of 8 beats (one cache line). Wrap burst not implemented. WLAST not implemented, use single-beat writes.	Supports AHB-Lite only.	Supports AHB-Lite only.	Supports AHB-Lite only.
Cache Configuration	4 KB, 2-way, 8-word line.	4 KB, 2-way, 8-word line.	No cache.	No cache.
Interface	AXI and local bus.	AHB-Lite for instruction and data. Optional interface are CXU-LI and RVFI.	AHB-Lite for instruction and data.	AHB-Lite for instruction and data.

The design recommendations are as follows:

- Use RX or MC core for applications requiring high-throughput memory reads. For example, code fetch from external memory.
- Avoid SM and NANO cores for burst-dependent designs. The cores are optimized for minimal resource usage, not memory bandwidth.
- Ensure cache is enabled and properly configured when leveraging burst reads on RX or MC cores.
- Plan for single-beat writes across all cores. Burst writes are not supported.

4. RISC-V Embedded Software Design Guidelines

This section describes the options when designing RISC-V embedded software systems using the Lattice Propel Software Development Kit (SDK).

4.1. Software Project Support

RISC-V NANO, SM, MC, and RX processors support bare metal software applications. The Board Support Package (BSP) provides drivers for processor startup initialization and accessing the IP in the system. When creating a new Lattice C/C++ Project in the Lattice Propel SDK, select **Hello World Project** for bare metal based applications.

The Lattice Propel SDK supports creating FreeRTOS based projects. FreeRTOS is available only when using the RISC-V RX processor. When creating a new Lattice C/C++ Project in the Lattice Propel SDK, select the **FreeRTOS Project**. For more information, refer to the FreeRTOS web page.

4.2. Board Support Package (BSP)

The Board Support Package (BSP) is generated when creating a Lattice C/C++ Project in the Lattice Propel SDK. The BSP provides the following:

- RISC-V processor start-up code and drivers for interrupt, exception, timer, cache, and watchdog
- Device drivers for IP in the system
- Platform header file that defines memory-mapped addresses, IP parameters, and C Macro

Table 4.1 lists the BSP components and the corresponding file hierarchy in the Lattice Propel SDK project.

Table 4.1. BSP Components and Hierarchy

Component	Hierarchy
RISC-V processor drivers	For RISC-V RX: *<Project Name>/src/bsp/driver/riscv_rtos/* For RISC-V MC: *<Project Name>/src/bsp/driver/riscv_mc/* For RISC-V SM: *<Project Name>/src/bsp/driver/riscv_sm/* For RISC-V NANO: *<Project Name>/src/bsp/driver/riscv_nano/*
Device drivers	*<Project Name>/src/bsp/<device name>* Example: *<Project Name>/src/bsp/driver/gpio/*
Platform header file	*<Project Name>/src/bsp/sys_platform.h*

4.2.1. Updating BSP

Update BSP when changes are made in the Lattice Propel Builder. This includes adding, removing, or upgrading IP, changing IP parameters, and changing the memory-mapped addresses. To update BSP, follow these steps:

1. Click **Design > Generate** in the Lattice Propel Builder to generate the latest system.
2. In the Lattice Propel SDK, under **Project Explorer** view, select the C/C++ project to update.
3. Click **Project > Update Lattice C/C++ Project…**.

   The **Update System and BSP** wizard opens as shown in Figure 4.1.

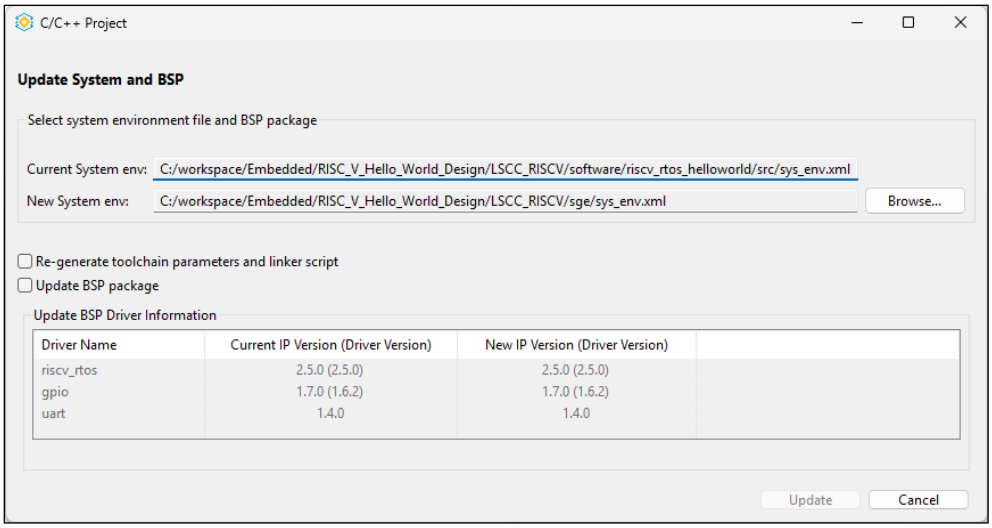

Figure 4.1. Update System and BSP Wizard

4. If your Lattice Propel Builder project has a different path as **Current System env**, click **Browse** to the latest Lattice Propel Builder project **sge/** directory and select the **sys_env.xml** file.
5. If you change the processor system memory, check **Re-generate toolchain parameters and linker script**. For example, if you change the processor connection to the program memory, update the connection in the linker script. Do not check this option if there is no change to maintain the current linker script.
6. Check **Update BSP package** to update the device drivers. The wizard shows the new driver version (when available) that for the updated BSP.
7. Click **Update** to make changes.

4.3. Software Compiler Optimization Options

Compiler optimization is a tool to improve performance and reduce code size with any modern CPU. Optimization is particularly important when dealing with Reduced Instruction Set Computer (RISC) architectures. RISC architectures such as RISC-V implicitly assume that the compiler solves structural issues that other architectures handle at the hardware level. Furthermore, well optimized code can make efficient use of the deep register files associated with RISC architectures, minimizing potentially costly accesses to main memory.

Optimization can significantly reduce instruction count, decrease memory footprint, and increase performance.

During the development phase, optimization may be counterproductive. Aggressive optimization increases compile times and slows down development. Optimization may also remove or reorder instructions, complicating debugging. For example, when single stepping through optimized code, execution may not be consistent with the high-level source (even though the code logically produces the correct result).

4.3.1. C/C++ Compiler Flow

The conventional process of compiling C/C++ source code into an executable binary image consists of four phases, as shown in Figure 4.2:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

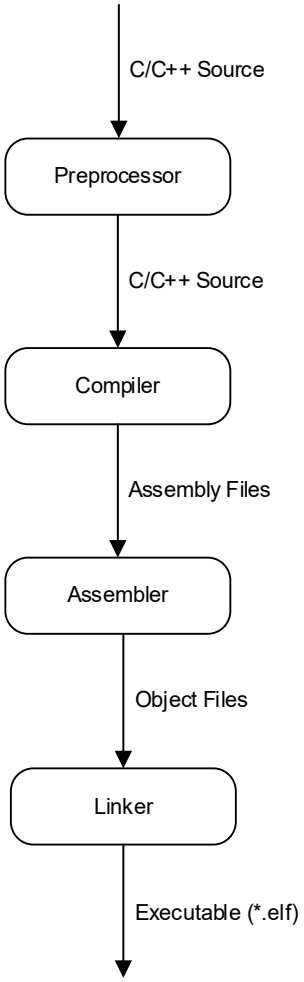

Figure 4.2. Phases of Compilation

Linking is the only phase that has a complete view of the entire program. Global optimizations across the whole program can only be performed by the linker.

Most optimizations are local optimizations performed by the compiler. Optimizations are performed on each source code module, one at a time.

4.3.2. Compiler Architecture

Compiling refers to converting high-level source into binary machine code. However, as shown in the C/C++ Compiler Flow section, the compiler performs only one part of the overall process.

The Lattice Propel SDK uses the GNU toolchain to build application images. The GNU compiler has three stages as shown in Figure 4.3.

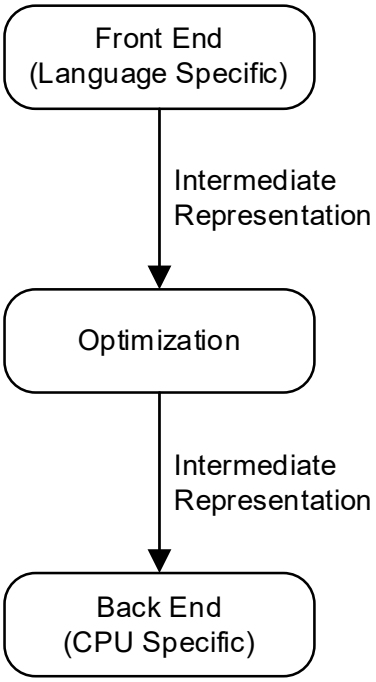

Figure 4.3. Generalized Compiler Architecture

The first stage which is the front end, is specific to a high-level language. The front end parses and analyzes high-level source code and converts the source code into a language independent intermediate representation that is passed to the next stage.

The second stage optimizes the intermediate representation that was produced by the front end. The nature of the optimization (for example performance versus code size) and the degree of effort are controlled by command line arguments to the compiler. When the second stage has completed optimization, a new, language independent, intermediate representation is passed to the final stage which is the back end.

A back end is specific to a CPU architecture. The primary function is to convert the intermediate representation produced by the optimization phase into instructions that can be executed on the target hardware. The output of the back end is assembly code that can be passed to the assembler, the next stage after the compiler in the build flow.

With this modular compiler architecture, the same optimization engine and target specific back end can support multiple high-level languages by using a different, language specific front end. Similarly, the same front-end and optimization engine can generate code for different variants of the same CPU architecture (for example 32-bit versus 64-bit) and different architectures (for example ARM versus RISC-V).

The front and back ends do little to optimize the code. If the compiler is invoked with optimization disabled, the output is a literal translation of the input source into assembly code. The output code has no logical reductions in instruction count and is not optimized for architectural features of the target CPU.

4.3.3. Optimization Levels

The GNU compiler supports many optimization algorithms. For a complete list of these algorithms, refer to the GCC, the GNU Compiler Collection web page. Most optimizations reduce the number of instructions in the compiled code which increases performance and reduces the program size in memory. However, some optimization algorithms increase code space to enhance performance or vice-versa.

Some optimization routines are iterative and can significantly increase compilation time.

The GNU compiler allows individual optimization algorithms to be invoked at the command line. The compiler also groups commonly used algorithms into different levels based on the desired tradeoffs. The levels and the corresponding command line arguments are shown in Table 4.2.

Table 4.2. GNU Compiler Optimization Levels and Command Line Arguments

gcc Command Line Argument	Description
-O0	Disables optimization
-O1	Runs optimizations that increase performance without increasing memory size and compilation time
-O2	Runs optimizations in -O1 and optimizations that increase compilation time
-O3	Runs optimizations in -O2 and adds optimizations that sacrifice code space for improved performance
-Ofast	Runs optimizations in -O3 and adds several optimizations that are not valid for C/C++ standards compliant code
-Os	Runs optimizations that reduce program size without impacting the performance
-Og	Runs subset of optimizations that do not alter execution order that is useful for debugging

4.3.4. C/C++ Compiler Optimization in the Lattice Propel Software

The Lattice Propel SDK invokes the GNU compiler to build applications. You can control the arguments that the Lattice Propel software sends to the compiler, including optimization level, via the GUI settings.

You can change the optimization level using the GUI in several ways. During project creation, you can change the optimization level in the C/C++ Compiler tab in the Lattice Toolchain Setting window, as shown in Figure 4.4.

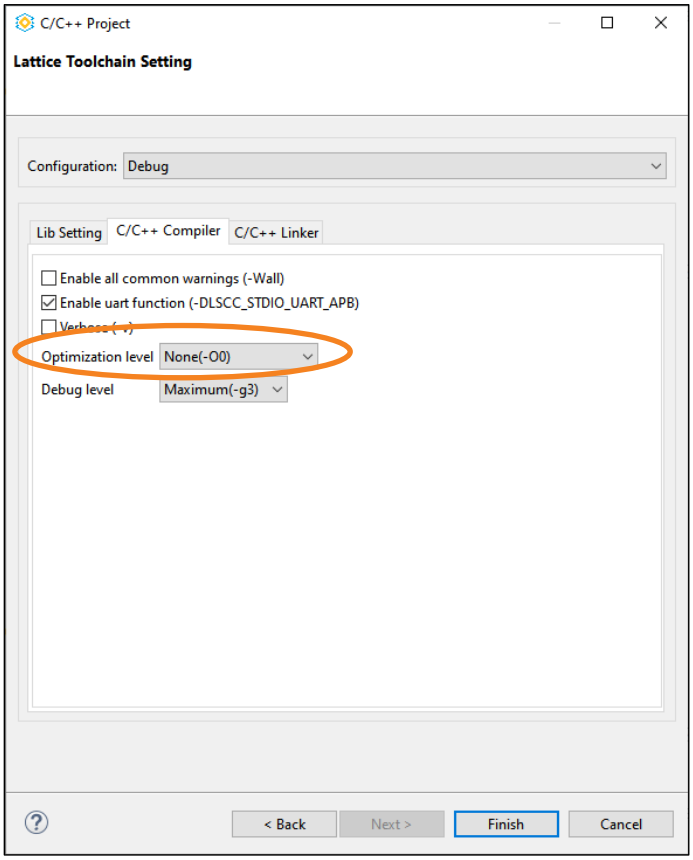

Figure 4.4. Optimization Setting during Project Creation

You can also change the optimization level after project creation in the Project Properties menu as shown in Figure 4.5.

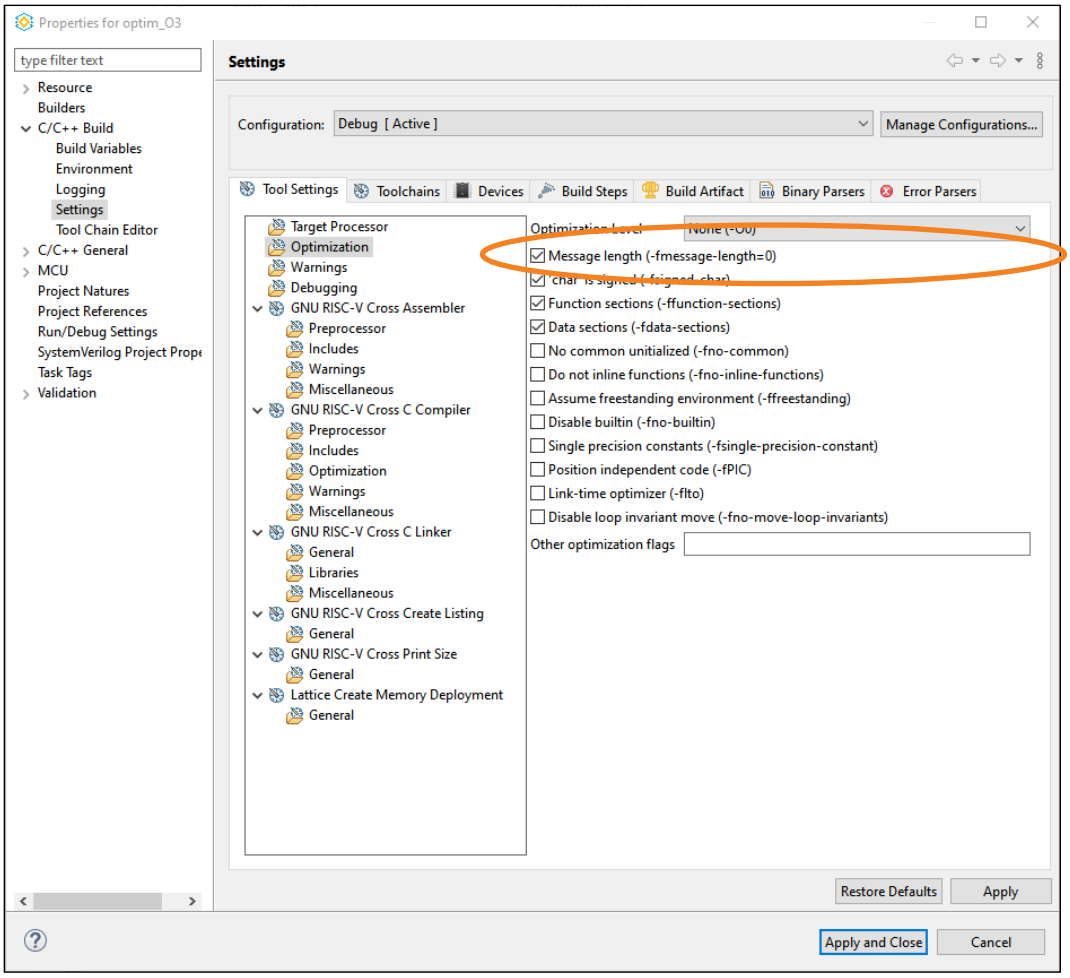

Figure 4.5. Changing Optimization Level from the Lattice Propel SDK Project Settings

The default Optimization Level is -O0 which turns off optimization. During the initial stages of a project, it is recommended to turn off optimization so that compile times remain short and instruction ordering matches that of the source code. For most applications, enable optimizations in the design cycle prior to the creation of a release candidate.

4.3.5. Fine Grain Control of Optimization

You can apply different levels of optimization to different parts of an embedded program using the GNU C/C++ compiler.

As shown in Figure 4.6, the `#pragma GCC optimize` command optimizes all subsequent code at the optimization level specified in the argument in double quotes.

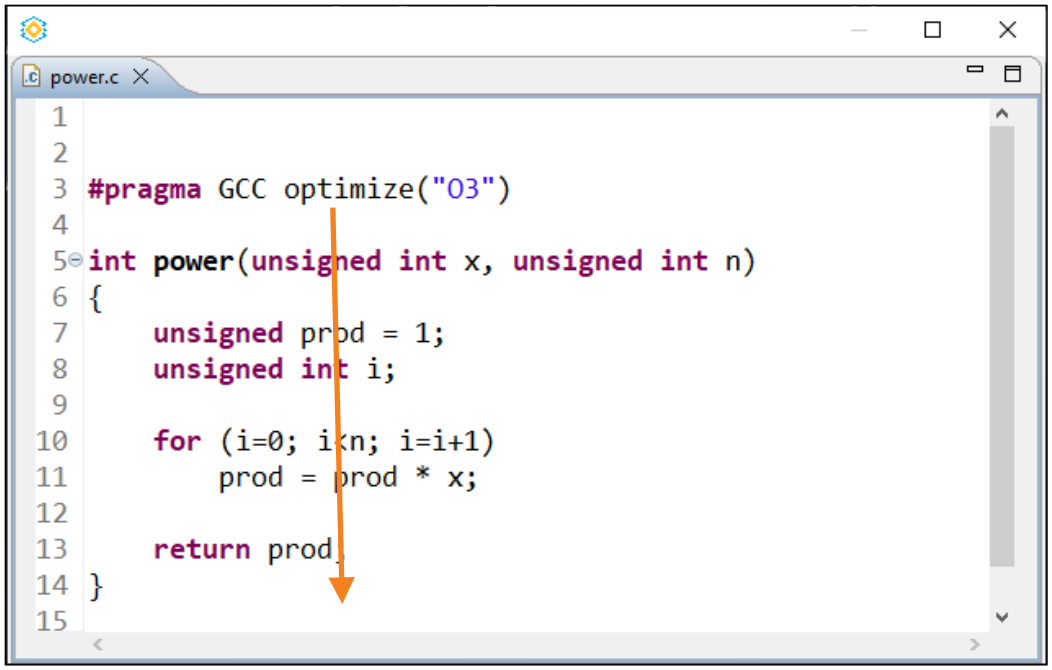

Figure 4.6. Using the #pragma Directive to Control Optimization Level

You can also apply optimization at the granularity of functions using the `__attribute__` in the GNU compiler. Different attributes can apply useful properties to functions, variables, type definitions, and optimization level. Figure 4.7 demonstrates the syntax for applying level 3 optimization to a function. The `__attribute__` keyword applies only to the specified function.

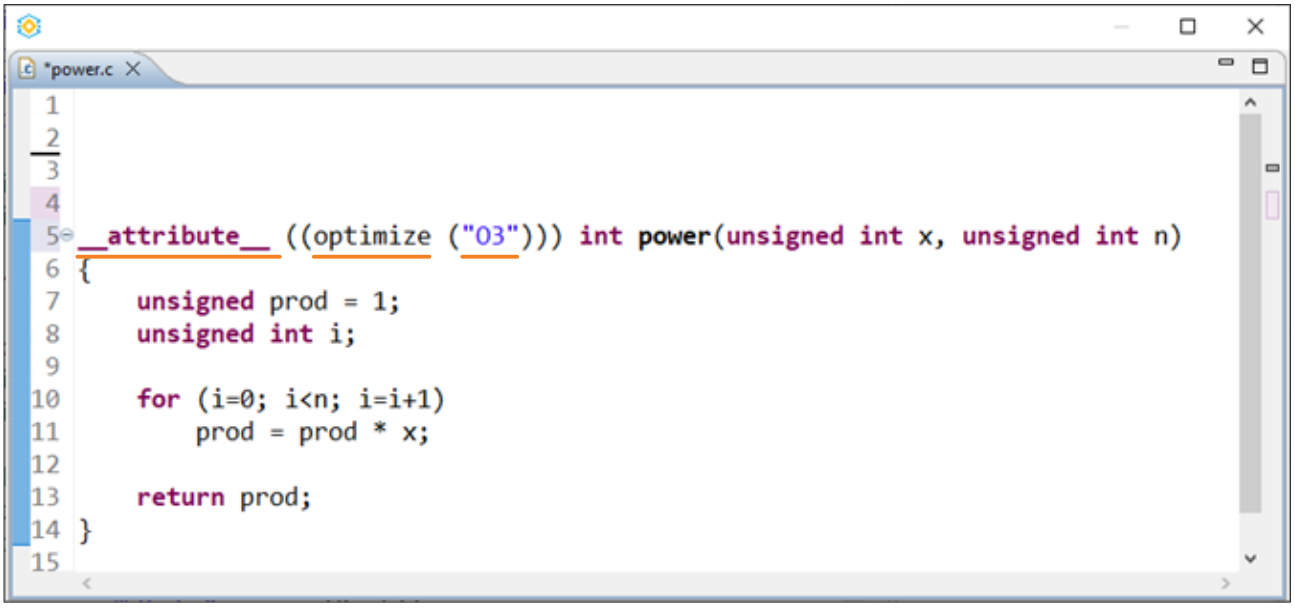

Figure 4.7. Applying an Optimize Attribute to a Function

4.3.6. Linker Relaxation

Some optimizations can only be applied globally. For RISC-V architecture, because of the design of the jump, load, and store instructions, linker relaxation optimization can only be performed once the whole program has been assembled and the locations of code and data are known.

The width of RISC-V instructions is equal to or less than the width of the address bus (32 bits). A RISC-V instruction cannot contain an immediate value that spans across the entire addressable memory space. Instead, RISC-V instructions compute a target address by adding a smaller immediate offset to a value contained in a register.

Loads and stores add a twelve-bit immediate field in the instruction to the contents of a register in the CPU register file. Jump instructions compute the target address by adding an immediate offset to either a register in the register file or to the Program Counter.

If the target address of a jump or a memory access is unknown at the time the instructions are emitted by the compiler, the compiler assumes the target address is outside the range of the offset provided by the immediate field. To cover the full range of potential target addresses, the compiler includes an instruction to load the higher order address bits of the target address into a register.

Because of code locality, the extra instruction is typically not necessary for most load, store, and jump instructions. However, it can only be determined at the linking stage whether a given operation requires a register to be loaded with the high order bits of its target address.

During linker relaxation, the linker scans the program as a whole and determines whether the target address of an operation is within the range of the immediate field from a register that holds a known value. The register for jump instructions can be the Program Counter. For load and store instructions, the register is in the register file and is designated as the Global Pointer. The Global Pointer is loaded during startup with an address that is near the location of most variables in memory.

If the linker detects that an operation can reach the target address without loading a new register value, the linker removes the extra register load instruction added by the compiler. This cuts in half, the number of instructions required for loads, stores and jumps. Linker relaxation results in significant gains in terms of both code size and performance.

Linker relaxation is enabled by default.

4.3.7. Link Time Optimization

Link time optimization is an option provided by the GNU C/C++ compiler and is a global optimization. This optimization forwards extra information to the linker and causes the linker to run optimization on the program.

Link time optimization may marginally improve performance. Link time optimization may increase code size.

Refer to the GNU GCC documentation for additional information.

Link time optimization is disabled by default.

4.4. Linker Script

Linking is the final step in translating a set of source files and libraries into an executable binary. Prior stages in the build process, which are the pre-processor, compiler, and assembler, produce a set of object files that the linker combines to form the complete program. Figure 4.8 shows the phases of compilation.

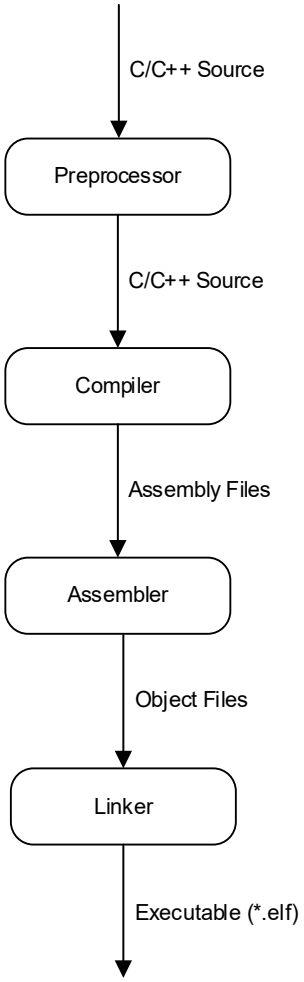

Figure 4.8. Phases of Compilation

You can control the GNU linker utility, ld, via command line arguments. For simple, natively compiled applications, the default settings are sufficient. More complex embedded applications require more control over the linking process. Control via command line becomes cumbersome and it is common practice to place linker arguments into a linker script file.

4.4.1. Linker Scripts in the Lattice Propel SDK

The Lattice Propel SDK automatically generates a linker script during the Lattice C/C++ Project creation. The autogenerated script is based on the memory configuration of the Lattice Propel Builder SoC design. This default script is suitable for simple designs. For more complex designs, add customizations to the default script.

The Lattice Propel SDK supports the following modifications to a linker script:

- GUI based linker script configuration
- Text based editing

4.4.1.1. GUI Based Linker Script Configuration

To open the GUI based linker script editor in the Lattice Propel SDK, double click the linker.ld file in the src directory of your Lattice C/C++ project. Figure 4.9 shows the Linker Configuration GUI.

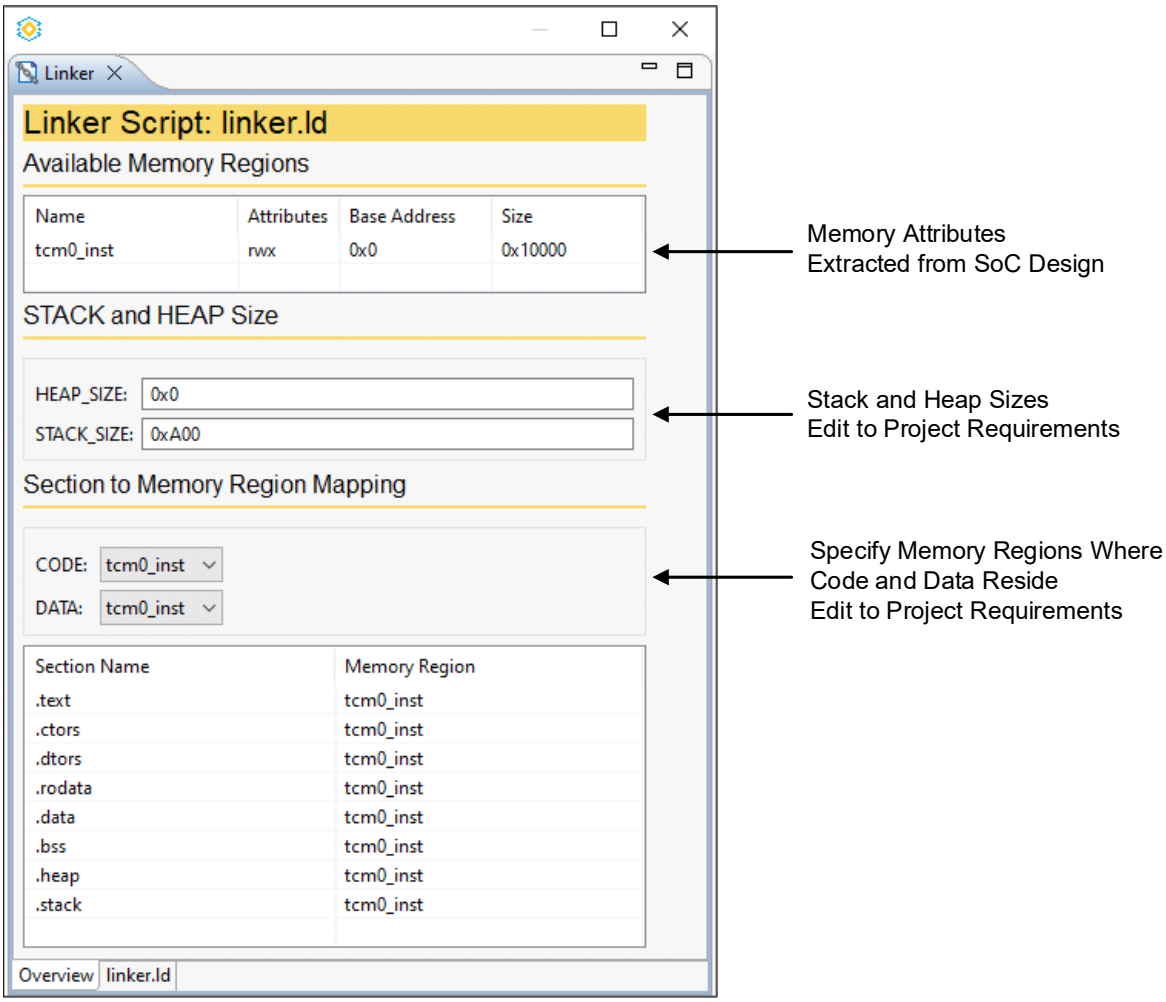

Figure 4.9. Lattice Propel SDK Linker Configuration GUI

The GUI displays the memory regions available in the SoC design on which the project is based. You can modify the stack and heap sizes by entering new values in the respective text boxes. For SoC designs with multiple memory regions, the GUI allows partitioning of code and data across different regions, enabling better optimization of memory usage and performance.

By default, the heap size is set to 0x0, indicating that dynamic memory allocation is disabled, while the stack size is set to 0xA00 (2560 bytes), which is suitable for simple applications. If your design uses dynamic memory functions such as malloc or new, it is recommended to set the heap size to a non-zero value, typically between 0x400 and 0x1000, depending on your application. For designs involving deep function calls, recursion, or large local variables, consider increasing the stack size to 0x1000 or higher to ensure reliable operation.

Always ensure that the total memory allocation, including code, data, heap, and stack, fits within the available memory resources of your target device. You can only make limited changes to the linking process using the Linker Configuration GUI. For more complex designs requiring more customizations, edit the text of the default linker script directly.

4.4.1.2. Text Based Linker Script Customization

To open the text based linker script editor, double click the linker.ld file in the src directory of your Lattice C/C++ project and click the linker.ld tab at the bottom of the pane as shown in Figure 4.10.

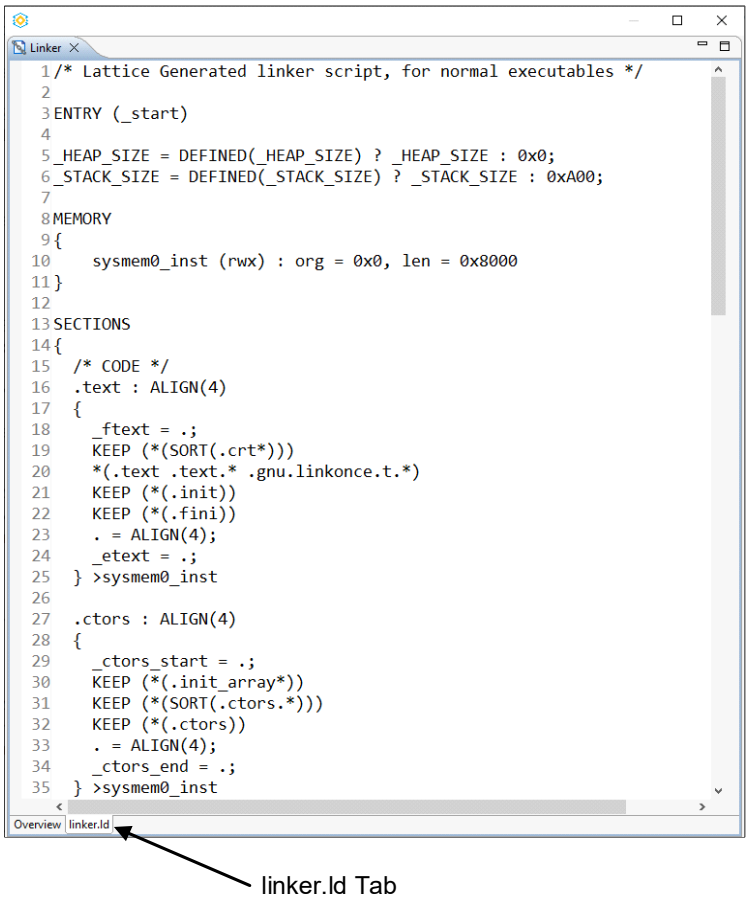

Figure 4.10. Lattice Propel SDK Linker Script Text Editor

4.4.2. Lattice Propel SDK Default Linker Script

GNU compatible linker scripts are written in the Linker Command Language. The default linker script provided by the Lattice Propel software consists of three commands in the following sequence:

1. ENTRY: Identifies the entry point, the first instruction to be executed in the program.
2. MEMORY: Specifies the different memory regions visible to the processor and the characteristics such as base address, size, and accessibility (read, write, execute).
3. SECTIONS: Specifies how the object file input sections are mapped to the ELF output sections.

Figure 4.11 shows the beginning of an autogenerated linker script for a system with two separate memories: one for storing instructions and one for storing data.

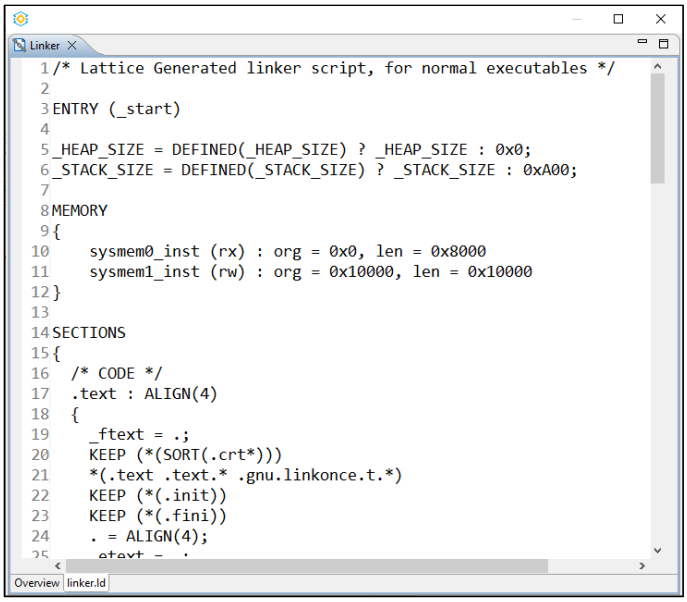

Figure 4.11. Autogenerated Linker Script for SoC System with Two Memory Regions

4.4.2.1. Linker Script ENTRY Command

The ENTRY command in the GNU Linker Command Language defines the first executable instruction in the output binary.

The ENTRY command takes a symbol name as an argument. In the Lattice BSPs for the bare metal and FreeRTOS execution environments, the startup code uses the symbol, "_start", to denote the first instruction. For example, Figure 4.12 shows the beginning of "crt0.S" for RISC-V MC where the first instruction, a jump, is assigned the "_start" symbol.

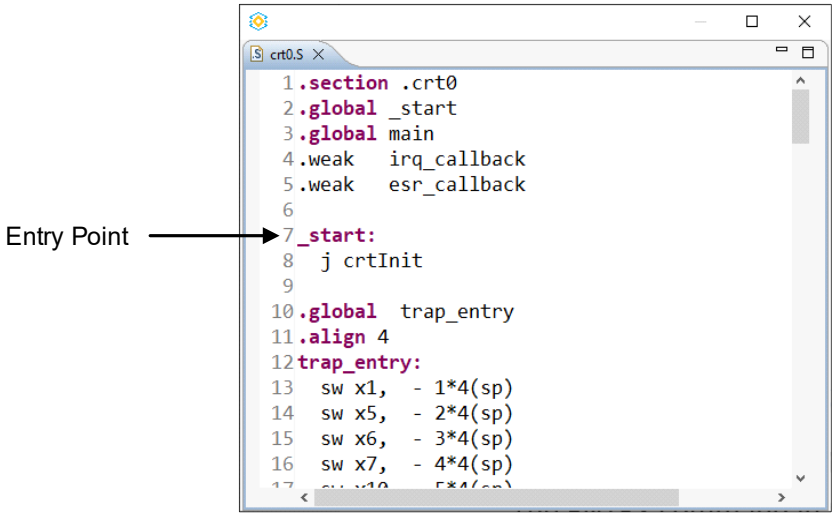

Figure 4.12. _start Entry Point in RISC-V MC BSP

4.4.2.2. Linker Script MEMORY Command

The MEMORY command in the GNU Linker Command Language informs the linker the memories available, memory locations in address space, memory sizes, and the operations the memories can support (for example read, write, or execute). The command also associates each memory block with a name that is used internally by the linker.

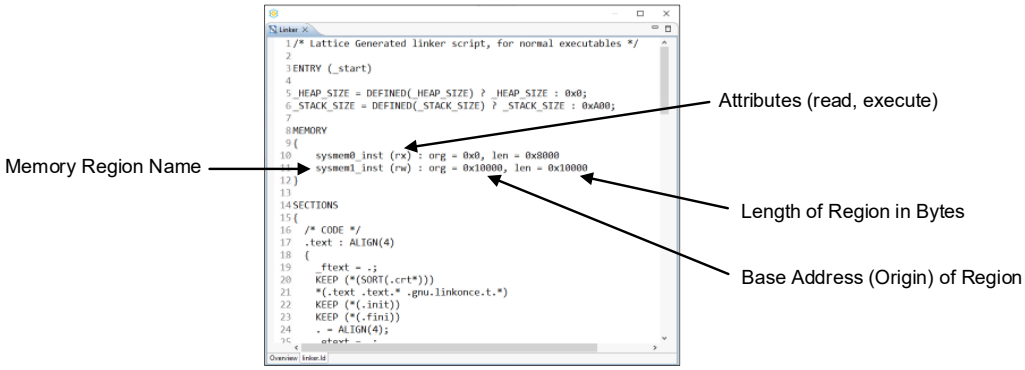

Figure 4.13. MEMORY Command Defining Two Memory Regions

In Figure 4.13, two memory regions are defined. The first, sysmem0_inst, is the instruction memory and is configured as read only in the Lattice Propel Builder SoC project. The autogenerated attribute field consists of only 'r' and 'x' for read and execute, and omits 'w' for write. Similarly, the sysmem1_inst memory is only connected to the data bus in the SoC design and the attributes are 'r' for read and 'w' for write while 'x' for execute is omitted.

Note: The Lattice Propel SDK debugger requires instruction memory to be read-writable via the RISC-V data bus. The debugger is not able to run on a Lattice SoC configured in Figure 4.13 because the program memory is not writable.

4.4.2.3. Linker Script SECTIONS Command

The SECTIONS command specifies how the parts (for example instructions, constants, variables, stack, and so on) of a program are arranged in memory.

The inputs to the linking process are a collection of object files. These files are a combination of pre-compiled libraries and high-level source modules of the program after being processed by the compiler and assembler.

The input object files are partitioned into sections. Some of the common sections are as follows:

- .text: program instructions
- .rodata: read only data (for example constant strings)
- .data: initialized global variables
- .bss: uninitialized global variables

The SECTIONS command defines the output sections that appears in the output *.elf file. The command also specifies the mapping of the input sections in the object files to the output sections.

The linker script that is automatically generated by the Lattice Propel SDK defines the following output sections:

- .text: program instructions
- .ctors/.dtors: C++ global constructor/destructor tables
- .rodata: read only data
- .data: initialized global variables
- .bss: uninitialized global variables
- .heap: pool of memory for dynamic allocation (for example malloc)
- .stack: program stack

The sections of the input object files and the file names (.text, .rodata, etc.) are defined by the GNU compiler and assembly tools. The output sections typically have the same names although this is not a requirement for the linker utility.

You can place the same type of program information in different sections. For critical code and non-critical code that are in the .text input sections, you can also place critical code in a cacheable memory region and place non-critical code using a different name in a non-cacheable memory region.

4.4.2.4. .text Output Section

Figure 4.14 shows the definition for the .text output section.

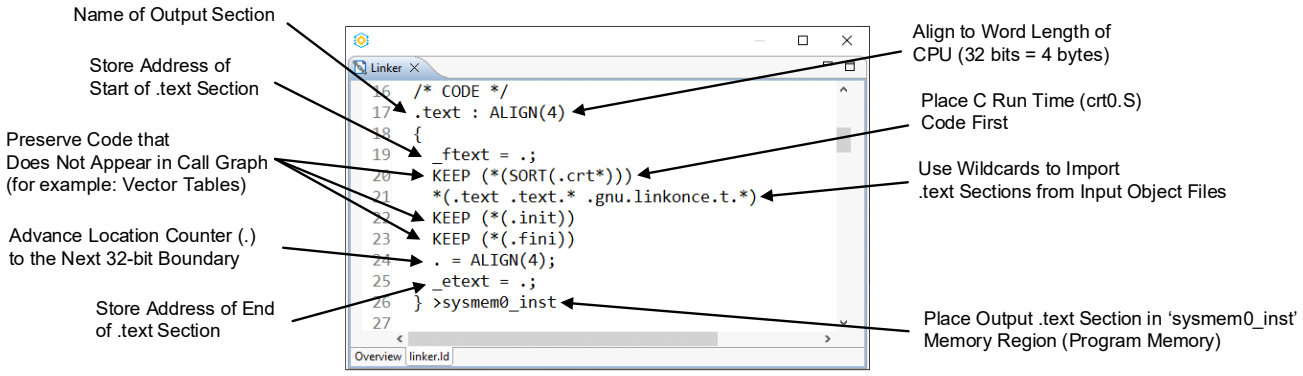

Figure 4.14. Autogenerated Linker Script .text Section

The output section declaration starts with the name of the section, followed by a colon and a call to the ALIGN function. ALIGN(4) forces the location counter to the next four byte or 32-bit boundary. This ensures that the instructions in the .text section are properly aligned for execution by the RISC-V processor.

The linker's location counter tracks the next location for code or data in memory. The period character, '.', is a built-in variable in the GNU Linker Command Language that provides access to the location counter.

The value of the location counter can be read into symbols. In the .text declaration, the '_ftext' and '_etext' symbols are assigned at the beginning and ending locations by reading the location counter before and after the input text sections are imported.

The key lines of this output section definition are in lines 20-23. These lines use wildcards ('*') to select input object files and sections. The input section names that match the specifiers are placed in the output .text section in the order in which they match.

The first match pulls the ".crt0" section into the .text output section. This section is defined in the crt0.S BSP file. As this section holds the C runtime startup code, this section is placed at the beginning of the program.

The KEEP keyword prevents the linker from pruning code and data which the call graph algorithms may mistakenly classify as unused. For example, the ".crt0" section includes interrupt vector tables that are not called explicitly, where KEEP ensures the linker does not remove the tables from the final executable.

The next matches are the .text sections from all input object files, which are the bulk of the program.

C++ constructs such as inline functions and virtual tables may cause code and data to be duplicated across multiple object files. The compiler places these functions and data into sections with names containing the "linkonce" pattern. When the linker encounters an input section name containing "linkonce", the linker adds that section to the output section for only one time. This prevents unnecessary code and data replication.

The ".init" and ".fini" contain library functions that execute global constructors at startup and global destructors when the program exits. These functions are called implicitly. The KEEP keyword ensures the functions are not removed from the .text output section.

Writing to the location counter built-in variable, '.', advances the location of the next placement. Figure 4.14 shows an example where ALIGN(4) is assigned to '.'. This assigned variable advances the location counter to the next 32-bit boundary.

The final line of the .text output definition directs the linker to place the section in the 'sysmem0_inst' memory region. This region is accessible via the SoC instruction bus and the base address is 0x00000000, which is the reset vector of the CPU.

4.4.2.5. .ctor and .dtor Sections

The .ctor and .dtor sections are mainly associated with C++ global constructors and destructors. Figure 4.15 shows the .ctors and .dtors output section definitions from the autogenerated linker script.

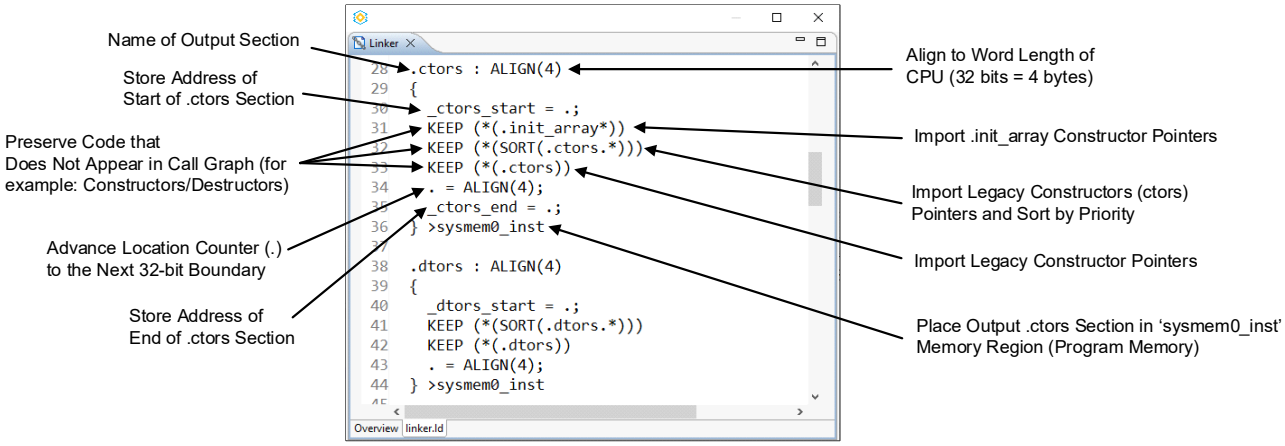

Figure 4.15. Definition for .ctors and .dtors Output Sections

The .ctors and .dtors sections are tables that hold pointers to the global constructors and destructors. As with other output sections, the .ctors and .dtors definitions start with the output section name followed by a call to the ALIGN function to ensure proper alignment to a 32-bit word boundary.

The linker script supports the following input section types that hold constructor and destructor pointer tables:

- The ".init_array" input sections are the modern format the GNU GCC uses to package global constructor and destructor pointer tables.
- The ".ctors" and ".dtors" input sections are included for backwards compatibility with the legacy code.

Although the constructor and destructor pointer tables are not executable code, the tables are not data and are closely related to the program. The .ctors and .dtors output sections are placed in the sysmem0_inst memory region with the .text section.

4.4.2.6. .rodata Section

The .rodata output section holds read only constant data. Figure 4.16 shows the output section definition for .rodata in the autogenerated linker script.

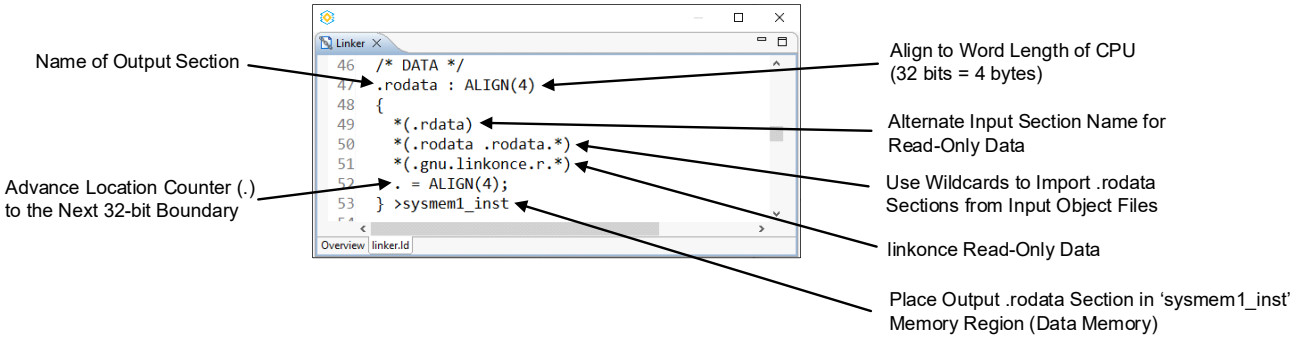

Figure 4.16. Definition for .rodata Output Section

The output section definition specifiers import ".rodata" input sections and input sections with an alternate name, ".rdata". The specifiers also match the name, ".gnu.linkonce.r", which can result from the inclusion of C++ constructs such as inline functions.

The final line of the .rodata output section definition places the section in the second memory region, sysmem1_inst, which is allocated for data.

4.4.2.7. .data Section

Figure 4.17 shows the definition for the ".data" output section.

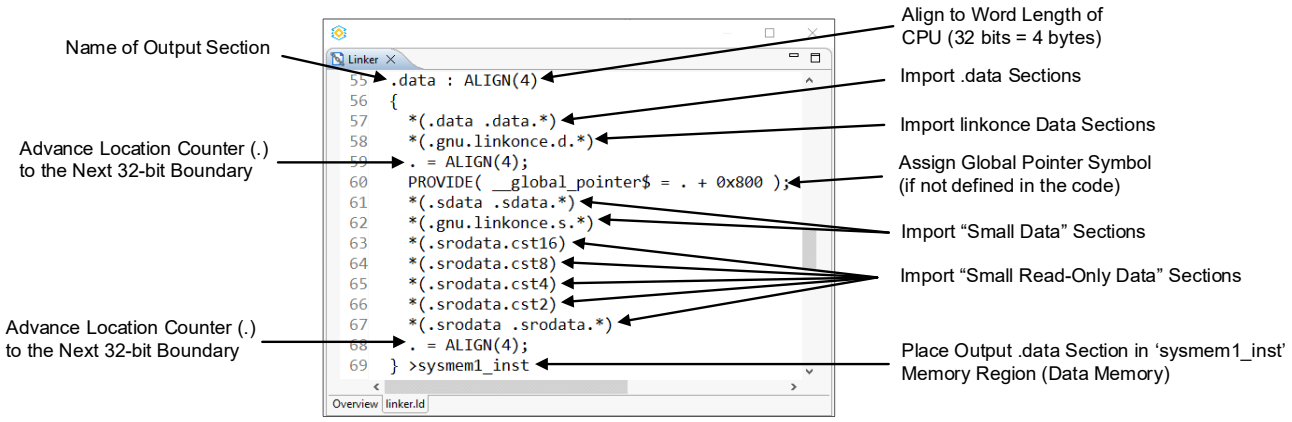

Figure 4.17. Definition for .data Output Section

The .data output section keeps global, initialized variables, particularly smaller variables, close to the address pointed at by the “global pointer” which is the “gp” register in the register file. This improves performance and code size via linker relaxation (see the Linker Relaxation section).

Figure 4.18 shows the encodings for the RISC-V load and store instructions, where a RISC-V core reads data from and writes data to the memory.

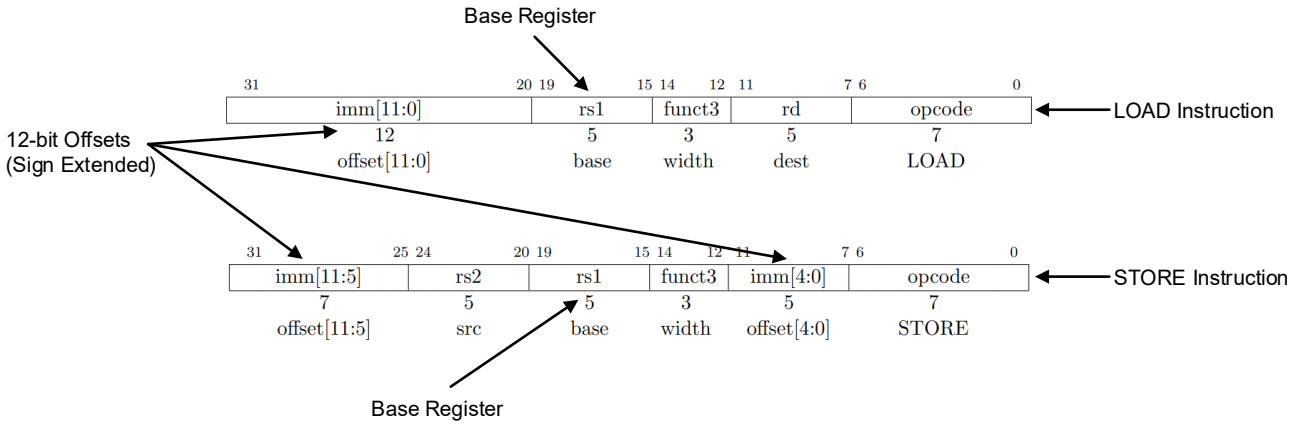

Figure 4.18. RISC-V Load and Store Instruction Encodings

The load and store instructions add a base address to a signed, 12-bit offset to specify the target address. The “rs1” field in the instruction indexes a register that holds the base address, while the immediate field encodes the offset. A naïve memory access requires two instructions: one to pre-load the base address into a register and a second to execute the load or store operation.

However, if a register already contains a value within the range of a signed, 12-bit immediate from the target address, the register pre-load is not required. The linker script places the smaller variables close to the global pointer register to ensure the memory access is within the 12-bit window of a known register value.

The output section definition imports the .data sections and linkonce data sections from the input object files. Depending on compiler settings, these sections are the larger data structures and arrays that are accessed via pointer. Linker relaxation is not able to enhance the performance for these sections.

After forcing alignment to the next 32-bit boundary, the script creates a __global_pointer$ symbol that points to the value of the location counter plus an offset of 0x800. The __global_pointer$ symbol is used in the startup code to initialize the “gp” register, a register in the register file.

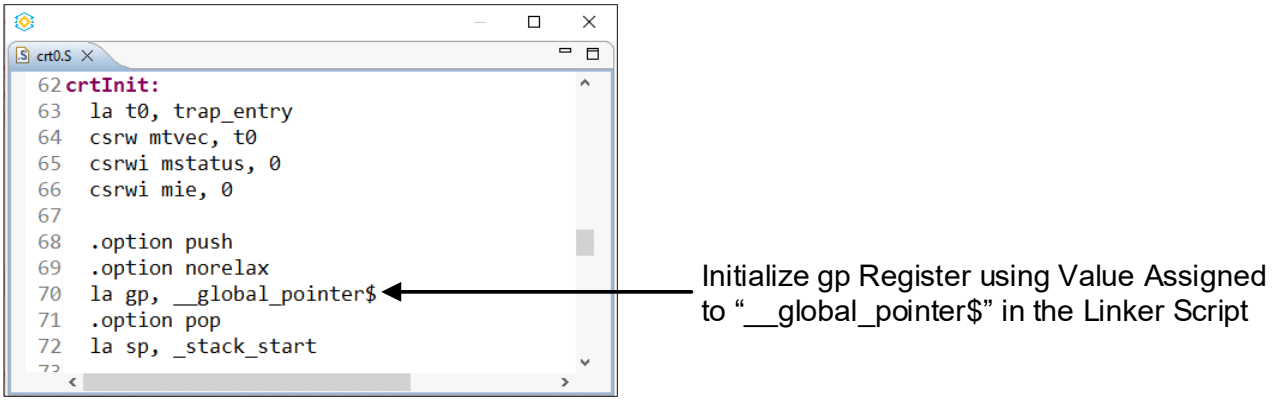

Figure 4.19. Startup Code Initializing gp (Global Pointer Register)

The "PROVIDE" keyword assigns the __global_pointer$ symbol if the symbol is not already given a value in the code.

After the assignment of the global pointer, the script imports the ".sdata" and ".srodata" sections which are the "small data" sections that the compiler creates based on a configurable threshold, as shown in Figure 4.20.

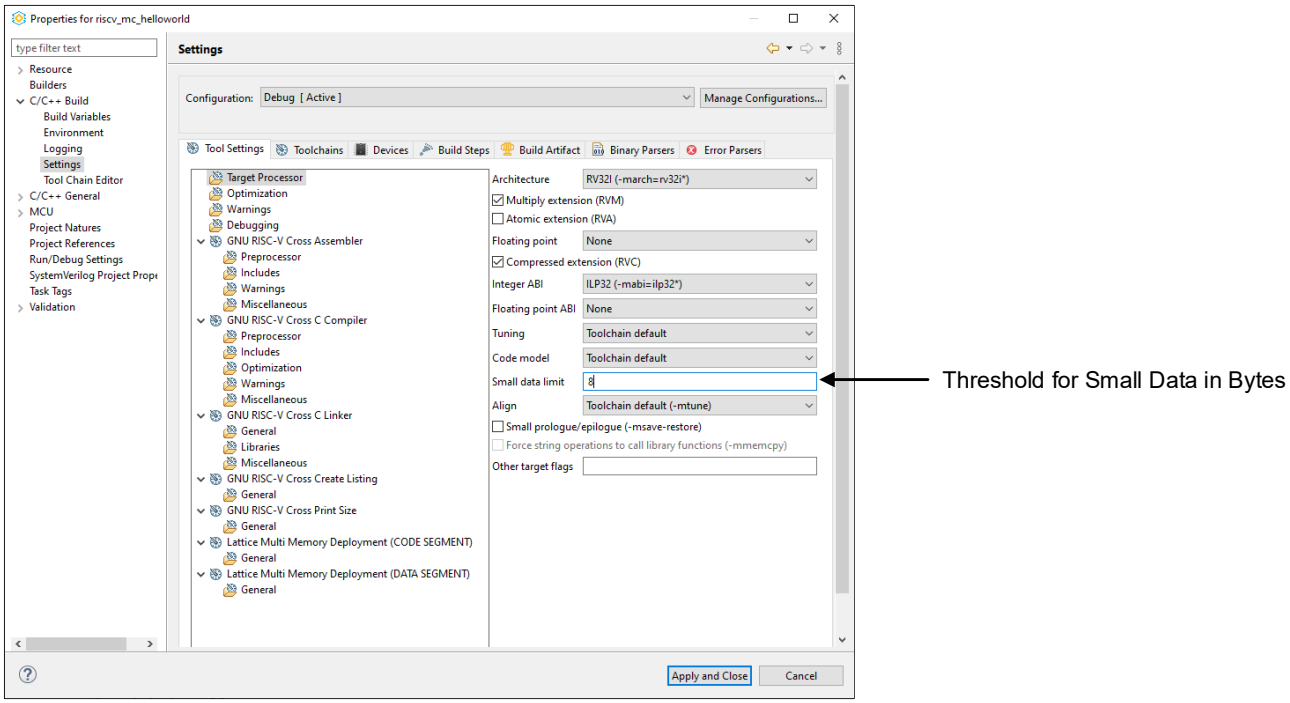

Figure 4.20. Configuration of the Small Data Limit in Lattice Propel SDK

The final line of the .data output section places the section in the sysmem1_inst memory region which is allocated for data storage.

4.4.2.8. .bss Section

The definition of the ".bss" output section is shown in Figure 4.21.

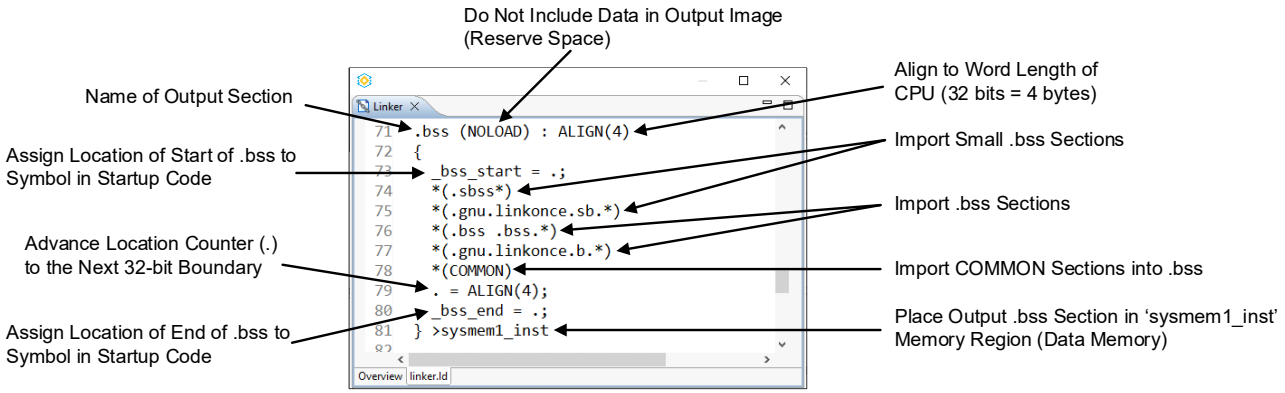

Figure 4.21. Definition for .bss Output Section

The .bss ("Block Start by Symbol") section reserves space for global and static variables that are not initialized by the source code.

In the C/C++ standards, such variables are required to be zeroed out. Instead of storing a bunch of zeros in the output ELF, the C startup code performs the zeroing of the .bss section at program launch.

The NOLOAD keyword instructs the linker to exclude the actual BSS data in the output image to avoid increasing the output binary image unnecessarily. For the C runtime code to initialize the memory associated with .bss, the linker points the C runtime code to the .bss location in the memory. The _bss_start symbol captures the value of the location counter at the beginning of the .bss section and the _bss_end symbol captures the value at the end of .bss section. C startup code starts and stops zero initializing the memory based on the _bss_start and _bss_end symbols.

Figure 4.22 shows the .bss initialization code in the crt0.S source file. The C run time implements a loop that uses the _bss_start and _bss_end symbols as initial and terminal values, respectively.

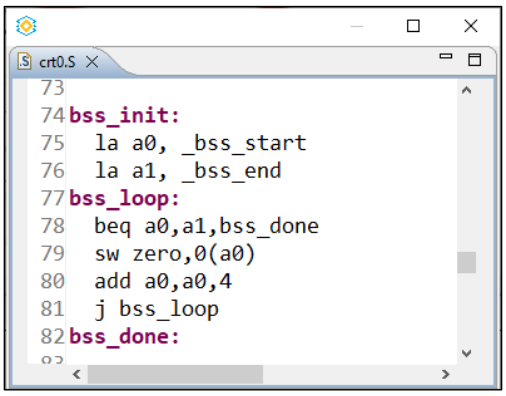

Figure 4.22. .bss Initialization Loop in crt0.S Source File

As .bss section contains variables that are placed in the memory region reserved for data, sysmem1_inst.

4.4.2.9. .heap Section

Figure 4.23 shows the output section definition for the ".heap" section.

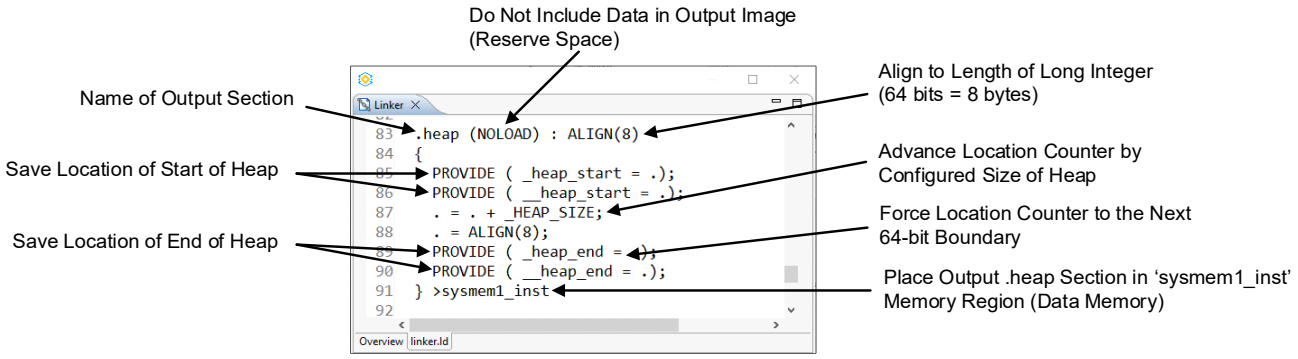

Figure 4.23. Definition for .heap Output Section

The heap is a pool of memory reserved for dynamic allocation. No variables are assigned to the heap at link time and there is nothing to initialize. The NOLOAD keyword instructs the linker to exclude initial values for heap in the output executable image.

To reserve the required space, the output section advances the location counter, '.', by the heap size as defined in the "_HEAP_SIZE" symbol near the top of the linker script.

As shown in Figure 4.24, _HEAP_SIZE and _STACK_SIZE symbols are defined using tertiary operators. If the symbols are already defined in the program, the existing value is used. If the symbols are not defined, the constant value from the Linker "Overview" GUI tab is used.

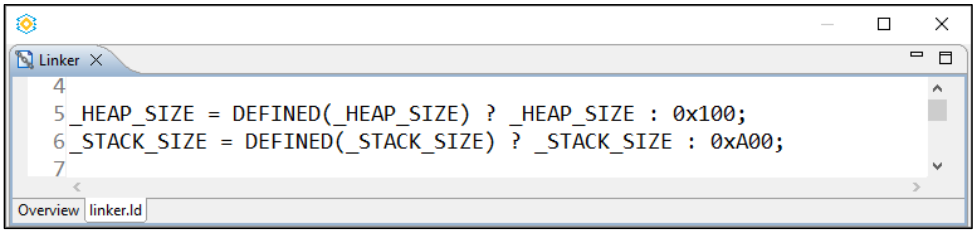

Figure 4.24. Defining Heap and Stack Size Symbol

The .heap output section definition also assigns multiple symbols to record the start and stop addresses of the heap, which is required for dynamic memory libraries such as malloc.

The last line of the .heap output section definition places the section in sysmem1_inst, the data memory region.

4.4.2.10. .stack Section

The definition of the ".stack" output section is shown in Figure 4.25.

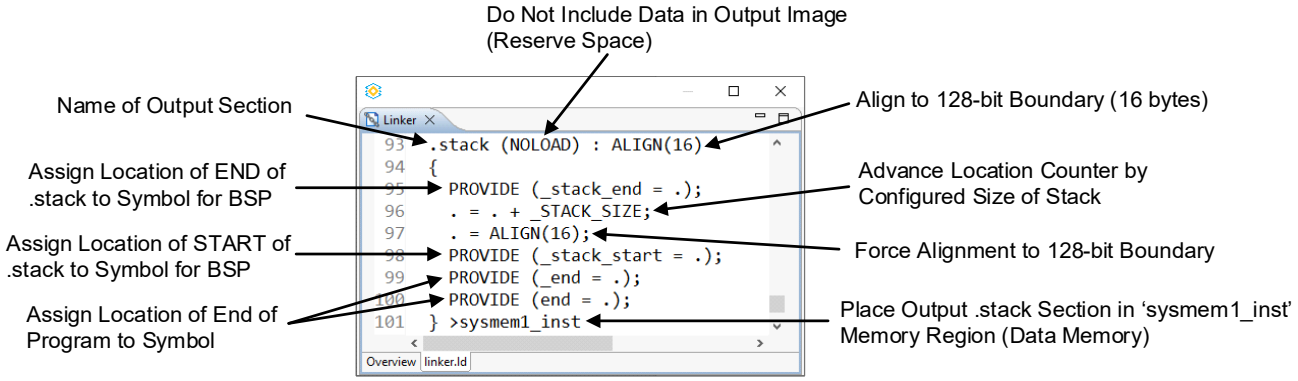

Figure 4.25. Definition for .stack Output Section

The .stack output section is aligned with a 128-bit boundary which is required by the RISC-V calling convention and is for the 'Q' extension which adds quad precision floating point. Note that 'Q' extension is not supported for Lattice RISC-V.

The .stack output section definition reserves space for stack. This section uses the NOLOAD keyword to inform the linker not to add the contents of the stack to the output ELF.

The linker script reserves memory for the stack by advancing the location counter by the amount of the _STACK_SIZE symbol. Like the _HEAP_SIZE symbol, _STACK_SIZE is defined near the beginning of the script using a tertiary expression that selects the value from the GUI if the symbol is not defined elsewhere.

The .stack output section definition also defines _stack_start and _stack_end symbols for use in the BSP. By convention, the RISC-V stack grows downward so the _stack_end symbol is assigned at the beginning of the section definition and _stack_start is defined at the end.

The stack is placed in the data memory region, sysmem1_inst.

4.5. Memory Initialization File Generation

Memory initialization files (.mem) are used to preload data into memory devices such as system memory and tightly coupled memory (TCM) blocks in FPGA-based SoC designs. These files are essential during simulation and synthesis to ensure that memory components contain the correct initial data, typically firmware or boot code, at system startup.

In the Lattice Propel SDK, the .mem file is generated during the build process using the following command sequence:

```
riscv-none-embed-objcopy -O binary --gap-fill 0 "<project_name>.elf" "<project_name>.bin"
```

```
srec_cat "<project_name>.bin" -Binary -byte-swap 4 -DISable Header -Output
"<project_name>.mem" -MEM 32
```

The first command converts the compiled ELF file into a raw binary format, while the second command formats the binary into a .mem file with 32-bit word width and appropriate byte ordering. The generated .mem file can be used to initialize memory blocks in FPGA.

4.5.1. Memory Initialization for Multi-Region SoC Design

In SoC designs where multiple memory devices are present, such as tightly coupled memory (TCM) and system memory blocks, each memory region may require the own initialization file to preload firmware or data during simulation or synthesis. This is important when different memory blocks serve distinct roles in the system.

The following are the use cases where multiple memory devices are implemented in a SoC design:

- Boot code in TCM, application in system memory
  The TCM stores boot code that must execute immediately after reset because of the low-latency access. Meanwhile, larger application code and data are stored in system memory, which may be slower but more spacious. This separation ensures fast boot-up and efficient memory usage.
- Real-time tasks in TCM, operating system in system memory
  In real-time embedded systems, critical tasks requiring deterministic timing are placed in TCM to avoid delays caused by memory access latency. The operating system and non-critical services are stored in system memory. This setup allows the system to meet real-time constraints while still supporting complex software stacks.
- Code and data in separate system memory devices
  In some designs, executable code and runtime data are placed in different system memory regions to improve memory organization and access efficiency. For example, one memory block stores the program instructions, while another handles dynamic data such as buffers, variables, or stack. This separation allows for better control over memory usage and simplifies debugging and performance tuning.

To generate a .mem file for each memory device, the binary output from the build process is cropped and offset according to the base address and size of each memory region using the following command format:

```
srec_cat <project_name>.bin -Binary -crop <base_address> <end_address> -offset -
<base_address> -byte-swap 4 -DISable Header -Output <memory_name>.mem -MEM 32
```

Figure 4.26 shows an example of a SoC design linker script with three memory devices.

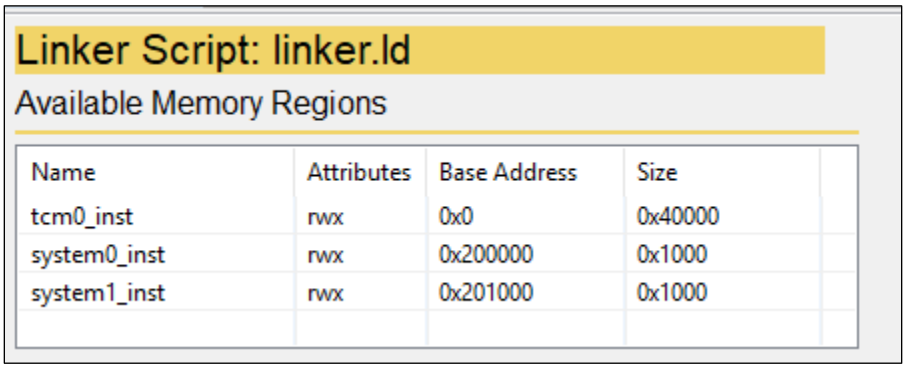

Figure 4.26. SoC Design with Multiple Memory Devices

The corresponding command is as follows:

```
srec_cat <project_name>.bin -Binary -crop 0x00000000 0x00040000 -offset -0x00000000 -byte-
swap 4 -DISable Header -Output tcm0_inst.mem -MEM 32; srec_cat <project_name>.bin -Binary -
crop 0x00200000 0x00201000 -offset -0x00200000 -byte-swap 4 -DISable Header -Output
sysmem0_inst.mem -MEM 32; srec_cat <project_name>.bin -Binary -crop 0x00201000 0x00202000 -
offset -0x00201000 -byte-swap 4 -DISable Header -Output sysmem1_inst.mem -MEM 32
```

The command consists of three separate srec_cat operations, each separated by a semicolon (;). These commands process different memory regions of the *<project_name>.bin* binary file and generate corresponding *.mem* files.

For the Lattice Propel SDK to include the commands in the build operation, the command needs to be added in the project **Properties > C/C++ Build > Settings > Build Steps > Post-build steps > Command** as shown in Figure 4.27. When the command is added, click **Apply and Close** for the new command to take effect in the next project build.

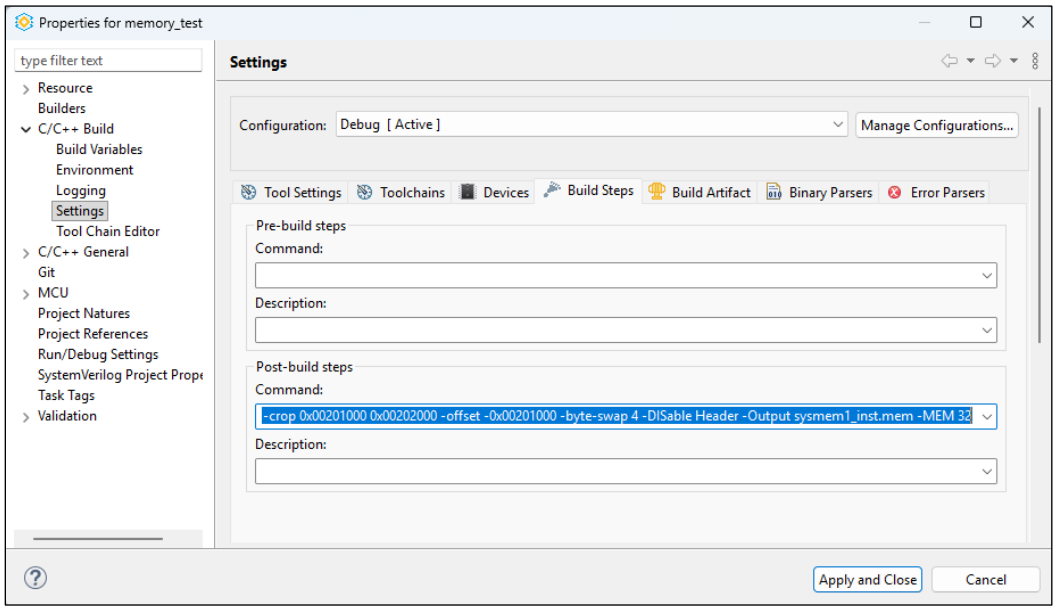

Figure 4.27. Adding Post-build Command for Generating Memory Initialization .mem File

To start project build, right-click on the project and click **Build Project**. The additional commands are executed at the end of the build operation as shown in Figure 4.28 and the generated .mem files are as shown in Figure 4.29.

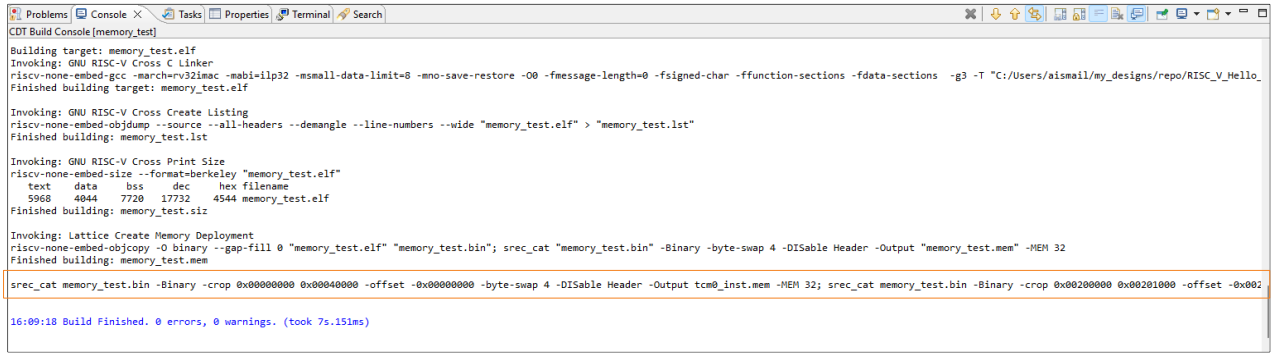

Figure 4.28. Post-build Command Console Log

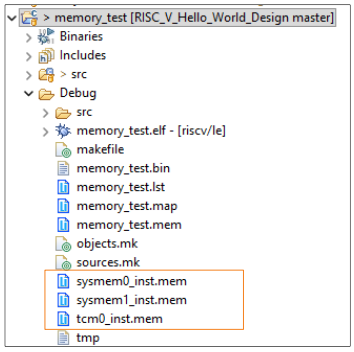

Figure 4.29. Generated .mem Files With Post-build Command

Each generated .mem file can be used as the memory initialization file for the corresponding memory device.

4.5.2. Adding Memory Initialization File into Design

The generated memory initialization file can be added in the design either in the Lattice Propel software project as permanent initialization content in the HDL module, or via the ECO Editor in the Lattice Radiant software project as temporary initialization content update into the bitstream file.

4.5.2.1. Adding Memory Initialization File in the Lattice Propel Builder

To add memory initialization file in the Lattice Propel Builder, follow these steps:

1. Open the Lattice Propel software project and double click the memory device to update the parameter.
2. Under the Memory section, select **Memory File** for memory initialization parameter, and select **hex** for **Memory File Format** parameter. Update the **Memory File** parameter with the file path of the generated .mem file.
3. Click **Generate** to complete the update.
4. Repeat the steps on each generated .mem files for other memory device.
5. Save the Lattice Propel software design to ensure all changes are saved permanently.

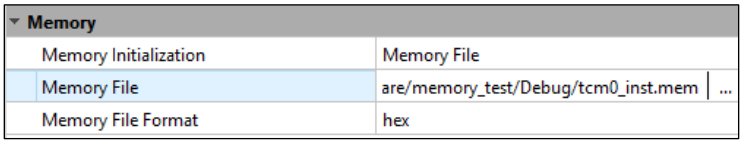

Figure 4.30. Memory Device Parameter Update in Lattice Propel Design

4.5.2.2. Adding Memory Initialization File into Radiant

To add memory initialization file in the Lattice Radiant software, follow these steps:

1. Open the Lattice Radiant software project. Ensure the project is compiled until the Place and Route Design stage, else proceed to run the compilation to complete this stage.
2. Click **Tools > ECO Editor**, select **Memory Initialization**. Look for the targeted memory instance, double click the input field of the Memory Initialization parameter.
3. In the **Memory Initialization Settings** window, select **Initialize by Memory File**. For File Format, select **Hexadecimal**, then update the Memory File parameter with the file path of the .mem file. Click **OK** to save the changes.

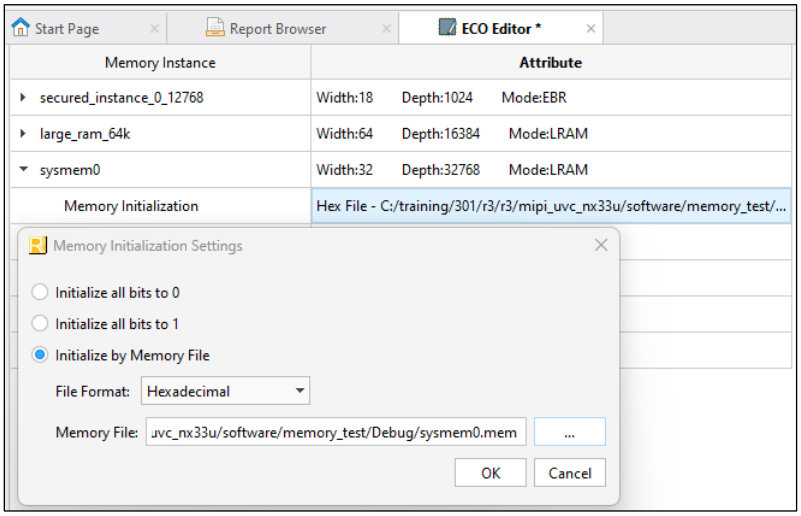

Figure 4.31. ECO Flow Update on Memory Instance Content Initialization

4. Repeat the steps for all .mem files.
5. Close and save the ECO Editor.
6. Click the stage of Export Files to generate the final bitstream file.

4.6. Interrupts and Exceptions

Interrupts are fundamental to embedded systems. An interrupt is essentially an unscheduled function call and is not synchronized to normal program flow. Using interrupts, an embedded processor responds to time critical events quicker and more deterministically than a loop that periodically polls for the condition. Most real time operating systems (RTOS) use preemptive multitasking and interrupts are integral to preemption. Interrupt hardware and software vary across different CPU architectures and execution environments. For example bare metal, schedulers, and real time operating systems. Understanding the CPU, interrupt controller, and BSP interrupt handler is key to effective and efficient development.

An exception is a type of event that disrupts the normal flow of program execution, typically triggered by the processor when the processor encounters an error or special condition during instruction execution, such as division by zero, invalid memory access, or illegal instructions. Unlike interrupts, which are usually triggered by external hardware events, exceptions are internally generated by the CPU. Handling exceptions is crucial for system stability and reliability, especially in embedded systems where fault tolerance is essential. Exception handling mechanisms vary across architectures, but the mechanisms generally involve saving the processor state, invoking a dedicated exception handler, and optionally recovering or terminating the affected process.

4.6.1. RISC-V Interrupt Architecture

The following is a brief overview of the RISC-V interrupt architecture. For detailed information on RISC-V interrupts, refer to the *RISC-V ISA specification, Volume II: RISC-V Privileged Architectures* in the RISC-V Specifications web page.

The RISC-V ISA groups interrupts and exceptions into a general category named “traps”. Certain RISC-V architectural features, including some Control and Status Registers (CSRs), are shared by interrupts and exceptions.

While interrupts are caused by events that are asynchronous to program execution, exceptions are synchronous. The CPU generates an exception as a direct result of executing a given instruction. This exception can be a result of an error such as a misaligned address or an illegal instruction. An exception may also be forced by design, such as when the ECALL instruction is executed.

When a trap, either an interrupt or an exception occurs, RISC-V transfers control to a trap handler firmware routine. The trap handler is part of the execution environment. For embedded systems, the trap handler usually comes with the board support package (BSP). The RISC-V interrupt logic obtains the memory address of the trap handler via a register that the execution environment loads during program startup.

Trap handler determines the cause of the trap and calls the appropriate service routine to respond to the interrupt or exception event. The RISC-V ISA defines the following methods to respond to the events:

- Directed mode causes the program counter to be loaded with the same address regardless of the cause of the trap.
- Vectored mode is for exceptions but uses a jump table for the different interrupt sources which can improve interrupt latency.

The RISC-V ISA comprehends three general interrupt sources: timer, software, and external.

- Timer interrupts are generated by a CPU’s integrated timer and are used for time-slicing in multi-tasking operating systems.
- Software interrupts are intended for inter-processor messaging, which allow one core or hart to interrupt another (or itself).
- External interrupts come from external sources, for example, the CPU’s peripherals.

For RISC-V implementations that support multiple privilege levels, interrupt sources are split into privilege mode-specific versions. For example, a RISC-V core that supports Machine and Supervisor privilege modes can have as many as 6 interrupt inputs: machine timer, machine software, machine external, supervisor timer, supervisor software, and supervisor external.

4.6.1.1. RISC-V Trap Handling CSRs

Table 4.3 list the key RISC-V control and status registers (CSRs) related to traps. These CSRs are for machine mode or supervisor level equivalents.

Table 4.3. Key RISC-V Trap Handling CSRs

CSR Name	Description
`mtvec`	Contains the vector base address for the trap handler. Controls the mode: directed or vectored.
`mcause`	Indicates the cause of the trap. Bit fields that indicate whether the trap is an interrupt or an exception and the type of interrupt or exception.
`mstatus`	Contains status and control bits including global interrupt enables and the status of the CPU core prior to the trap.
`mepc`	The value of the program counter is copied to the mepc register when a trap occurs. The contents of the mepc register are written back to the program counter when returning from the trap.
`mscratch`	Scratchpad register. Useful for operating systems in storing an index into a task context table.

For complete details about the RISC-V CSRs, refer to the RISC-V ISA specification, *Volume II: RISC-V Privileged Architectures* in the RISC-V Specifications web page.

4.6.2. Lattice RISC-V Interrupt Controller Hardware

An embedded system may need to support multiple external interrupts. As a RISC-V core supports only one external interrupt input per privilege level, interrupt controller hardware must be added to RISC-V based designs to aggregate external interrupts into a single signal.

The external interrupt controller for the Lattice RISC-V CPUs varies depending on the model, as shown in Table 4.4.

Table 4.4. Lattice RISC-V External Interrupt Controllers by Model

RISC-V Model	External Interrupt Controller
RX	Platform Level Interrupt Controller (PLIC)
MC	Programmable Interrupt Controller (PIC)
SM	Programmable Interrupt Controller (PIC)
NANO	Lightweight Interrupt Merge Controller (LWIMC)

4.6.2.1. Programmable Interrupt Controller (PIC)

The Programmable Interrupt Controller (PIC) is a proprietary Lattice IP. The PIC is used on the RISC-V MC and SM processors. Figure 4.32 shows a block diagram of the PIC.

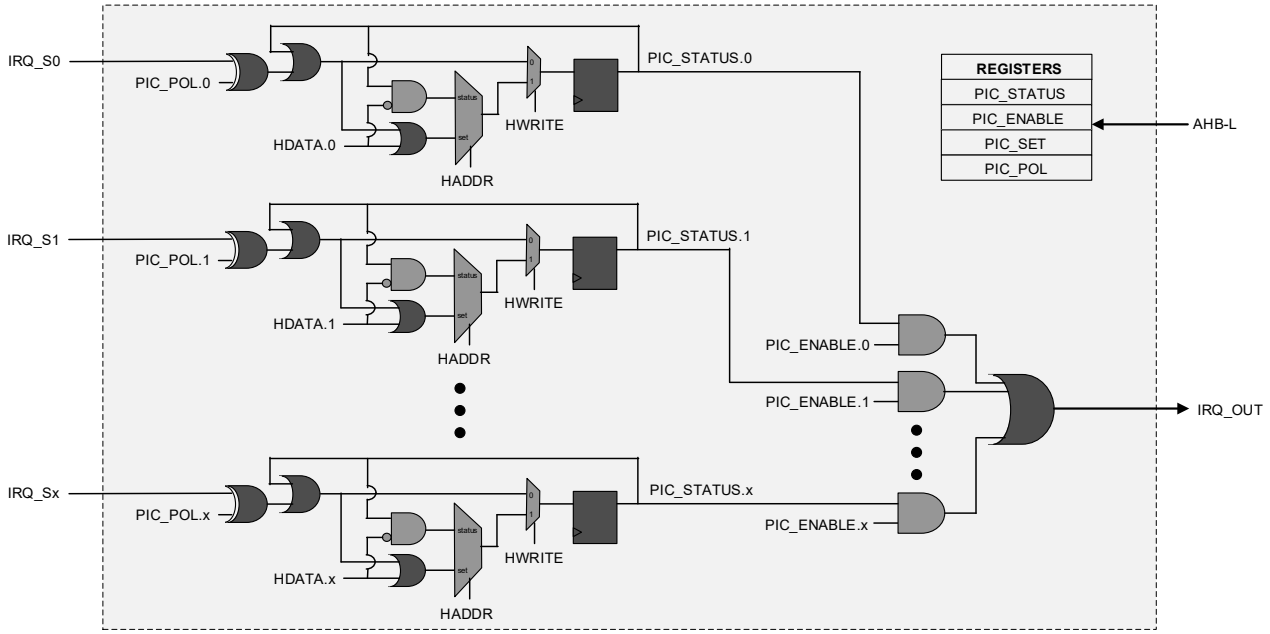

Figure 4.32. Programmable Interrupt Controller (PIC)

When enabled, the PIC supports 16 external interrupt inputs for SM and 32 for MC. The PIC has an AHB-Lite control interface consisting of four registers: PIC_STATUS, PIC_ENABLE, PIC_SET, and PIC_POL (polarity).

The interrupt channels for PIC are not complicated. The flip-flop holds the current pending interrupt state of the channel and the state can be read via the PIC_STATUS register. The input is inverted or not inverted based on the value of the respective bit in the PIC_POL register. The pending state can be cleared by an AHB-L write to the PIC_STATUS register address with a value of one in the respective data bit. The firmware can force a channel into the pending state by writing a one to the corresponding bit of the PIC_SET register. The bits in the PIC_ENABLE register control whether a pending interrupt on the respective channel causes the IRQ output to be asserted. All channels are identical and the channels are independent from one another.

In the RISC-V MC and SM processors, IRQ_OUT output from the PIC drives the external machine interrupt input to the CPU core.

4.6.2.2. Platform Level Interrupt Controller (PLIC)

The Platform Level Interrupt Controller (PLIC) is defined by the RISC-V Task Group. The Lattice RISC-V RX processor uses the PLIC as the interrupt controller. Figure 4.33 shows the block diagram of the PLIC from the specification.

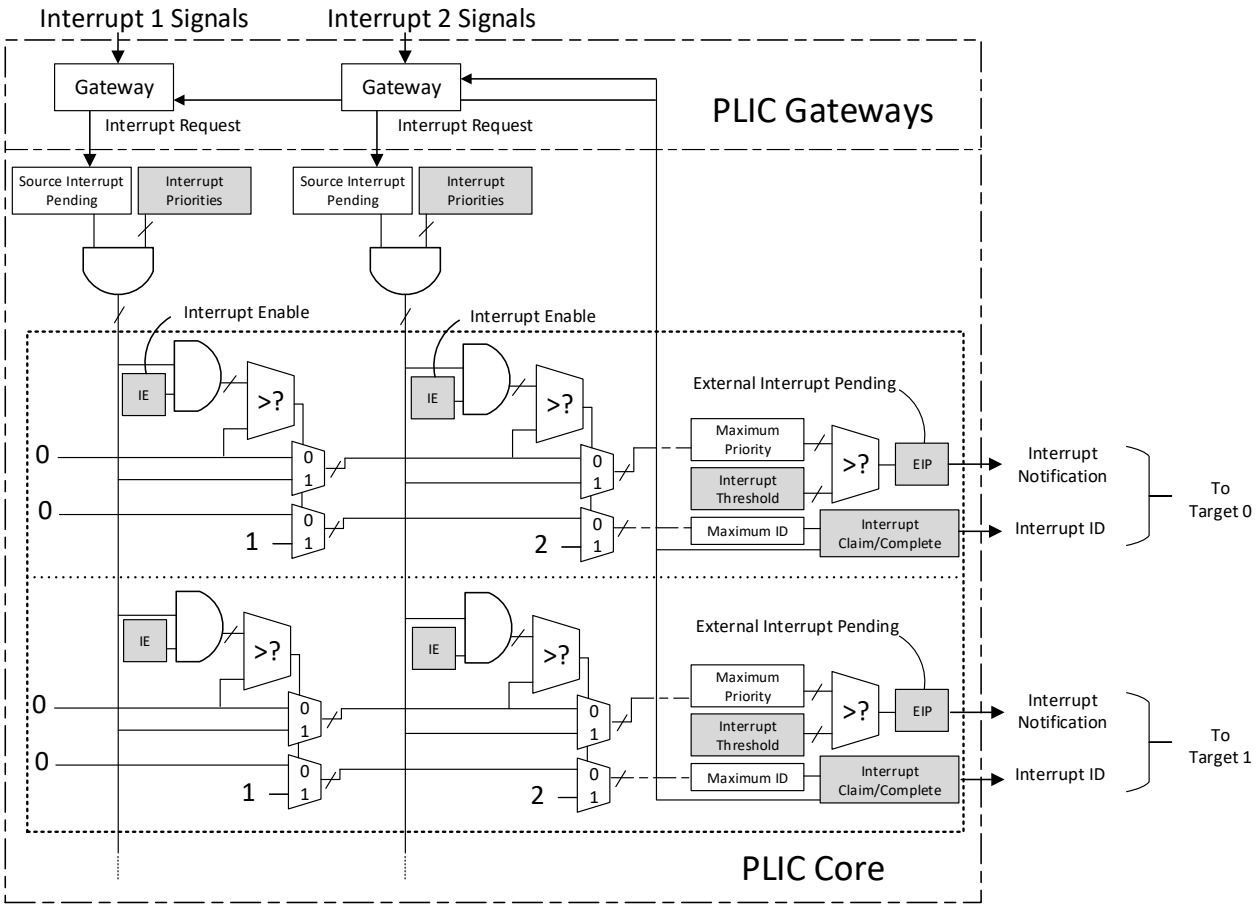

Figure 4.33. Platform Level Interrupt Controller (PLIC) Block Diagram

PLIC has up to 1024 interrupt sources. The Lattice RISC-V RX implementation supports only up to 32 interrupt sources. The first interrupt, interrupt zero (at the left of Figure 4.33), is reserved and not functional.

The PLIC has two outputs, one for machine privilege level external interrupts and one for supervisor level external interrupts.

The gateways at the top of the block diagram recognize and queue up interrupts. The PLIC gateways in the RISC-V RX support only level sensitive, high true, interrupts. When a gateway detects an interrupt assertion, the gateway sets the corresponding interrupt pending bit, IP.

The interrupt priority of an interrupt is a programmable register. When the interrupt pending bit, IP, is set, the interrupt priority bit field is forwarded to the PLIC core. If the interrupt enable bit, IE, is set for machine or supervisor outputs, the interrupt priority continues to be forwarded towards the respective output.

If a second, lower numbered interrupt is also pending at the same time, the two priorities are compared. The larger of the two continues to be forwarded, along with an interrupt ID field. If both interrupts have the same priorities, the interrupt with the lower interrupt ID value continues to be forwarded.

The winning priority is then compared with an interrupt threshold that is programmable on a per output basis. If the priority of the interrupt is greater than the threshold, the external interrupt pending bit, EIP, is set and the interrupt is asserted to the appropriate privilege level of the CPU.

The PLIC employs a Claim/Complete protocol to service and retire interrupts. To initiate a claim, the firmware performs a read of the claim/complete register of the PLIC. The PLIC returns the ID of the highest priority interrupt that is pending. If no

interrupt is pending, the PLIC returns a zero, the ID of the reserved, non-functional interrupt input. During the claim, the PLIC also clears the claimed interrupt's interrupt pending bit, IP.

Although the claimed interrupt's IP bit is cleared, the interrupt cannot occur again until the PLIC receives a completion from the firmware. The firmware performs a completion by writing the interrupt ID back to the claim/complete register. The completion is the final step in handling the interrupt.

It is important to note that the PLIC does not forward a new interrupt request from the same source until the PLIC receives this completion signal. This behavior ensures that only one interrupt per source is pending at any time. If the interrupt source reasserts before the completion is issued, the new interrupt may be missed. Therefore, firmware must ensure timely completion and proper handling of the interrupt source, especially for level-sensitive interrupts, which must remain asserted until the condition is cleared and the completion is written.

For complete technical details on the PLIC, see the specification in the RISC-V Platform-Level Interrupt Controller Specification web page and the RISC-V RX CPU IP – Lattice Propel Builder 2025.2 User Guide (FPGA-IPUG-02302).

4.6.2.3. Lightweight Interrupt Merge Controller (LWIMC)

Lightweight Interrupt Merge Controller (LWIMC) is a proprietary Lattice IP and is used in RISC-V NANO variant. Figure 4.34 shows the block diagram illustration of the LWIMC.

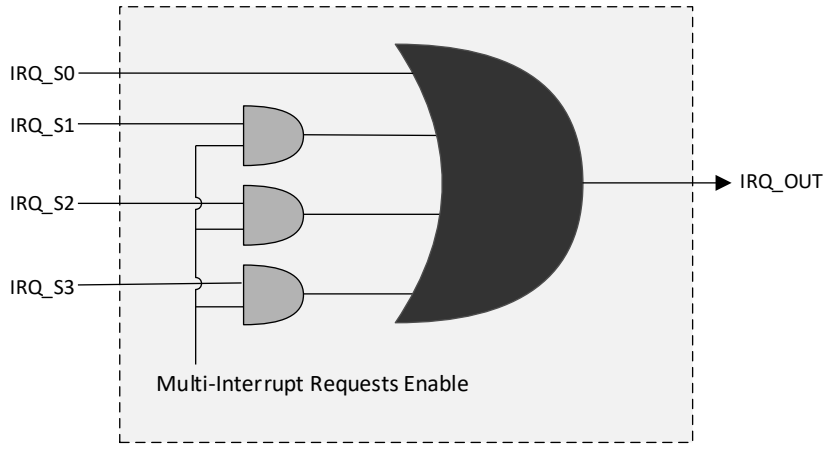

Figure 4.34. Lightweight Interrupt Merge Controller

Key information about the interrupt controller is as follows:

- Multi-interrupt requests:
  The interrupt controller can handle multiple interrupt requests when the *Multi-Interrupt Requests Enable* attribute is enabled. This attribute allows aggregation of up to four peripheral interrupts. If disabled, only one peripheral interrupt is supported.
- Interrupt inputs , IRQ_Sx:
  The interrupt ports are named IRQ_Sx (where x ranges from 0 to 3) and are used for external peripheral interrupts.
- Interrupt output, IRQ_OUT:
  IRQ_OUT output from the LWIMC drives the machine external interrupt input to the CPU core.

4.6.3. Lattice RISC-V Exceptions

In RISC-V architecture, exceptions are indicated and handled through a mechanism involving trap handling. When an exception occurs, such as an illegal instruction, misaligned memory access, or environment call, the processor transfers control to a trap handler by jumping to an address specified in the mtvec (machine trap-vector) register. The cause of the exception is recorded in the mcause register, which identifies the type of exception, while the mepc register stores the program counter of the instruction that caused the exception. This allows the handler to analyze the exception and potentially resume execution or take corrective action. RISC-V modular and extensible design makes the exception handling both flexible and efficient for embedded and real-time systems.

Table 4.5 shows the exceptions supported by each RISC-V variant. Each exception code in RISC-V represents a specific type of fault encountered during program execution. The codes are described as follows:

- 0 — Instruction address misaligned: Target address for a branch or jump is not aligned to instruction size.
- 1 — Instruction access fault: Instruction memory access failed because of protection or hardware error.
- 2 — Illegal instruction: Opcode and encoding are not supported by the configured ISA or explicitly illegal.
- 4 — Load address misaligned: Address not divisible by access size. For example, 32-bit load from non-word-aligned address.
- 5 — Load access fault: Load from memory failed because of access violation or memory error.
- 7 — Write access fault: Failure when writing to memory because of protection violations. For example, writing to read-only memory.

Table 4.5. Supported Exceptions for Each RISC-V Variant

Exception Code	Exception Name	RX (Lite/Balanced/Advanced)	MC	SM	NANO
0	Instruction Address Misaligned	Supported	Supported	Supported	Supported
1	Instruction Access Fault	Supported	Supported	Supported	—
2	Illegal Instruction	Supported	Supported	Supported	—
4	Load Address Misaligned	Supported	Supported	Supported	—
5	Load Access Fault	Supported	Supported	Supported	—
7	Write Access Fault	Supported	Supported	—	—

4.6.4. Lattice RISC-V Trap Handlers

The trap handling firmware that is part of the Lattice RISC-V BSPs is written in assembly. The trap handlers for RISC-V MC/SM/NANO and RISC-V RX are different, primarily because of the different execution environments for the processors: bare metal for RISC-V MC/SM/NANO and FreeRTOS for RISC-V RX.

Both trap handler routines perform the same general operations as follows:

1. Save the CPUs state, such as register file and key CSRs.
2. Determine the source of the trap: interrupt or exception.
3. Call the appropriate handler routine.
4. Restore the CPUs state. The state may be different following a task switch.
5. Execute the appropriate return instruction to resume the program.

4.6.4.1. Bare Metal Trap Handler

Figure 4.35 shows the bare metal trap handler that is part of the RISC-V MC/SM/NANO BSP.

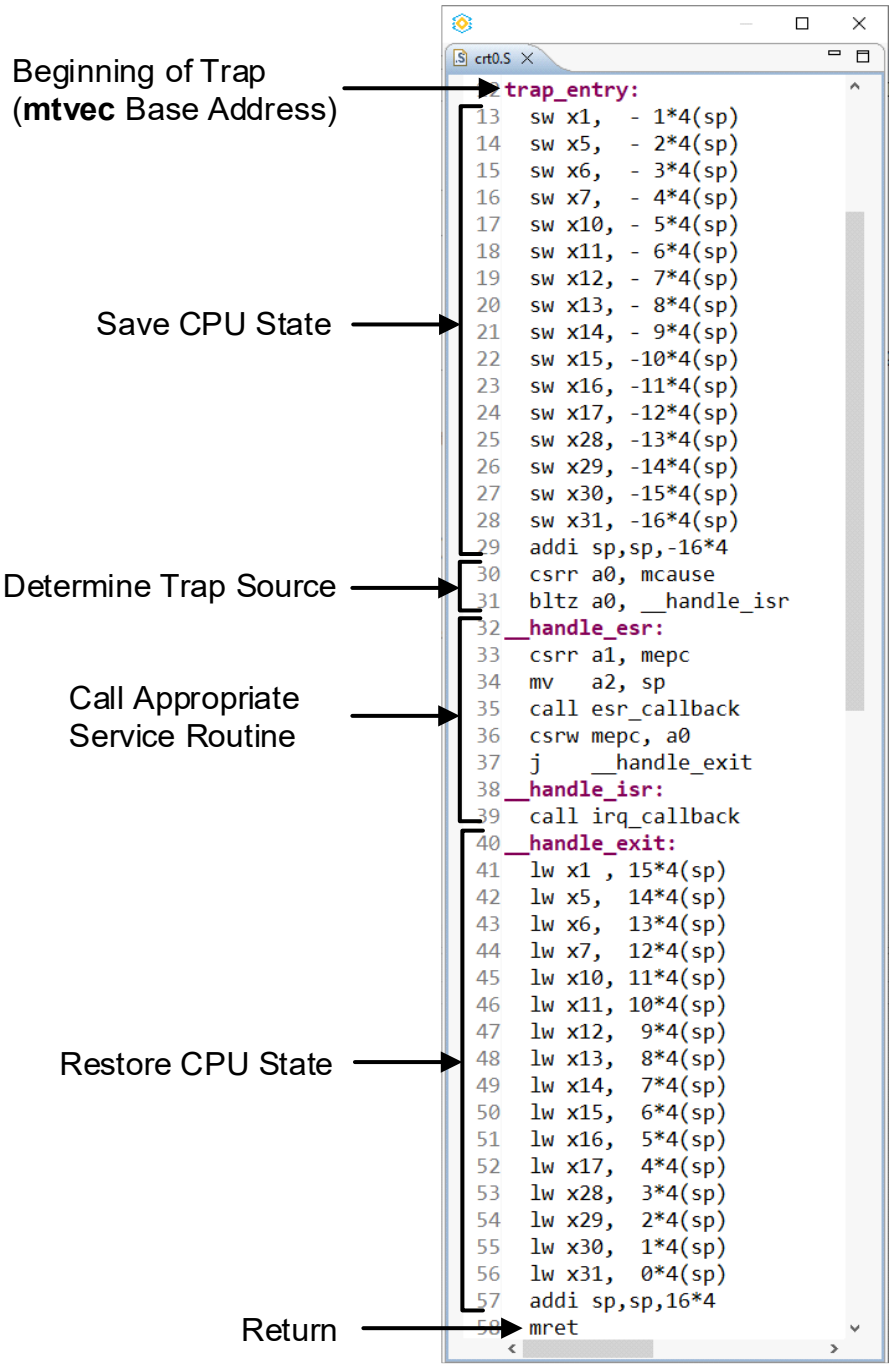

Figure 4.35. Bare Metal Trap Handler for RISC-V MC and SM

The trap_entry symbol marks the beginning of the trap handler routine. The startup code uses this symbol to initialize the **mtvec** CSR. This assignment causes the CPU hardware to jump to the trap handler when an interrupt or exception occurs.

RISC-V hardware does not automatically push the CPU state to the stack. The firmware stores the state of the machine during an interrupt by using a series of `sw` (store word) instructions and the `addi` operation. The sixteen registers stored are used by the bare metal execution environment. The integer addition operation updates the stack pointer by subtracting sixty-four bytes (16 words), the amount that the stack has grown downward in memory.

After the CPU state is saved, the trap handler retrieves the value in the **mcause** CSR and loads the value into a register, a0, for comparison. The most significant bit of **mcause** indicates whether the trap is caused by an interrupt or an exception. As the most significant bit is also the sign bit, a branch if less than zero (`bltz`) instruction can be used to test the value of the bit and branch to the appropriate routine to handle the trap.

For an interrupt, the C function, **irq_callback()** is called.

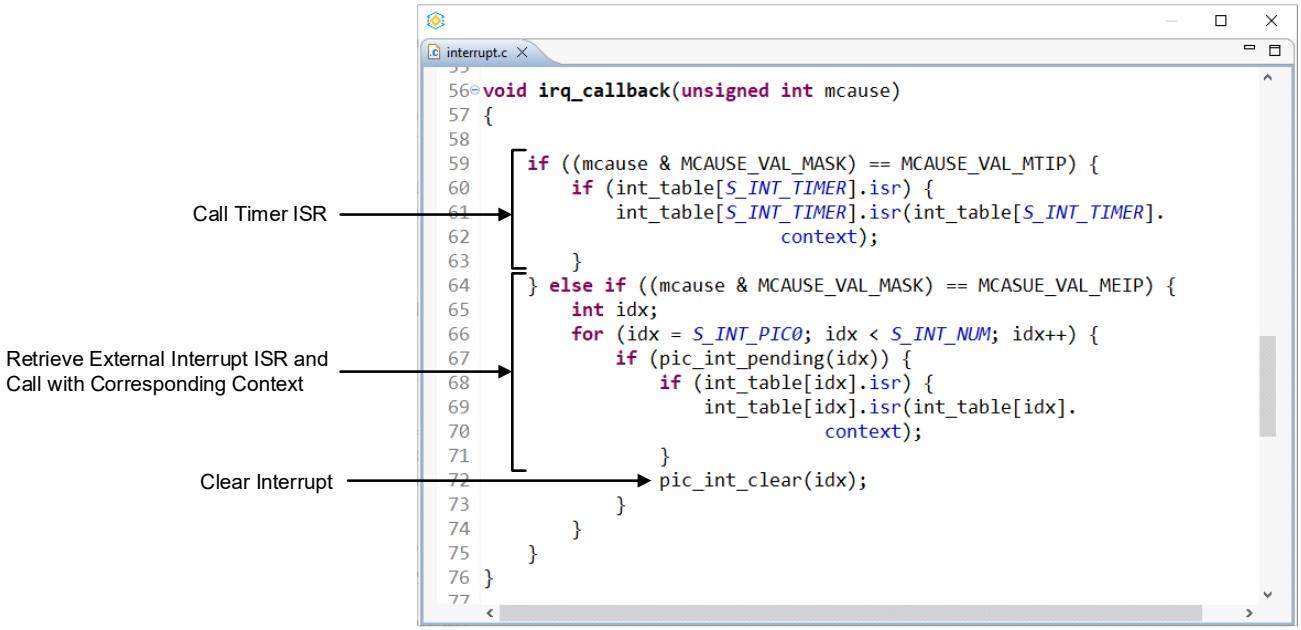

Figure 4.36. Bare Metal BSP irq_callback( )

The **irq_callback()** function relies on a global, array-based table named **int_table**. The table stores the following pointers:

- Function pointers to the interrupt service routines for the timer and the external interrupts.
- A pointer to a context data structure for each interrupt input. The pointer is passed to the interrupt service routine when the routine is called.

The context data allows the service routine function to avoid hard coding key parameters such as the base address of an associated peripheral. It also permits a single function be used as the interrupt service routine for multiple instances of the same type of peripheral.

The initialization routine of the timer configures the table entry for the timer ISR. Table entries for the external interrupts are initialized by API calls to the **pic_isr_register()** function.

When called, the **irq_callback()** function determines the source of the interrupt and looks up the corresponding table entry in **int_table**. The function calls the ISR function using the context data. Once the ISR finishes processing the interrupt and returns, the interrupt is cleared in the PIC (if the interrupt was an external interrupt) and program flow returns to the trap handler where the CPU state is restored and normal program execution resumes.

4.6.4.2. Lattice FreeRTOS Trap Handler

In Lattice RISC-V systems running FreeRTOS, trap handling is managed by the default FreeRTOS trap handler function `freertos_risc_v_trap_handler`. This function is responsible for handling all exceptions and interrupts during FreeRTOS execution.

As shown in Figure 4.37, to register the trap handler, the `freertos_init()` function must be called in the `main()` function before invoking `vTaskStartScheduler()`. This initialization step configures the mtvec register to point to `freertos_risc_v_trap_handler`, ensuring that all traps are correctly routed to FreeRTOS.

By default, the Lattice FreeRTOS operates in directed mode interrupt for interrupt handling. In this mode, all traps are directed to a single entry point defined by the mtvec register. Vectored mode interrupt is not supported in Lattice FreeRTOS implementation and must not be used.

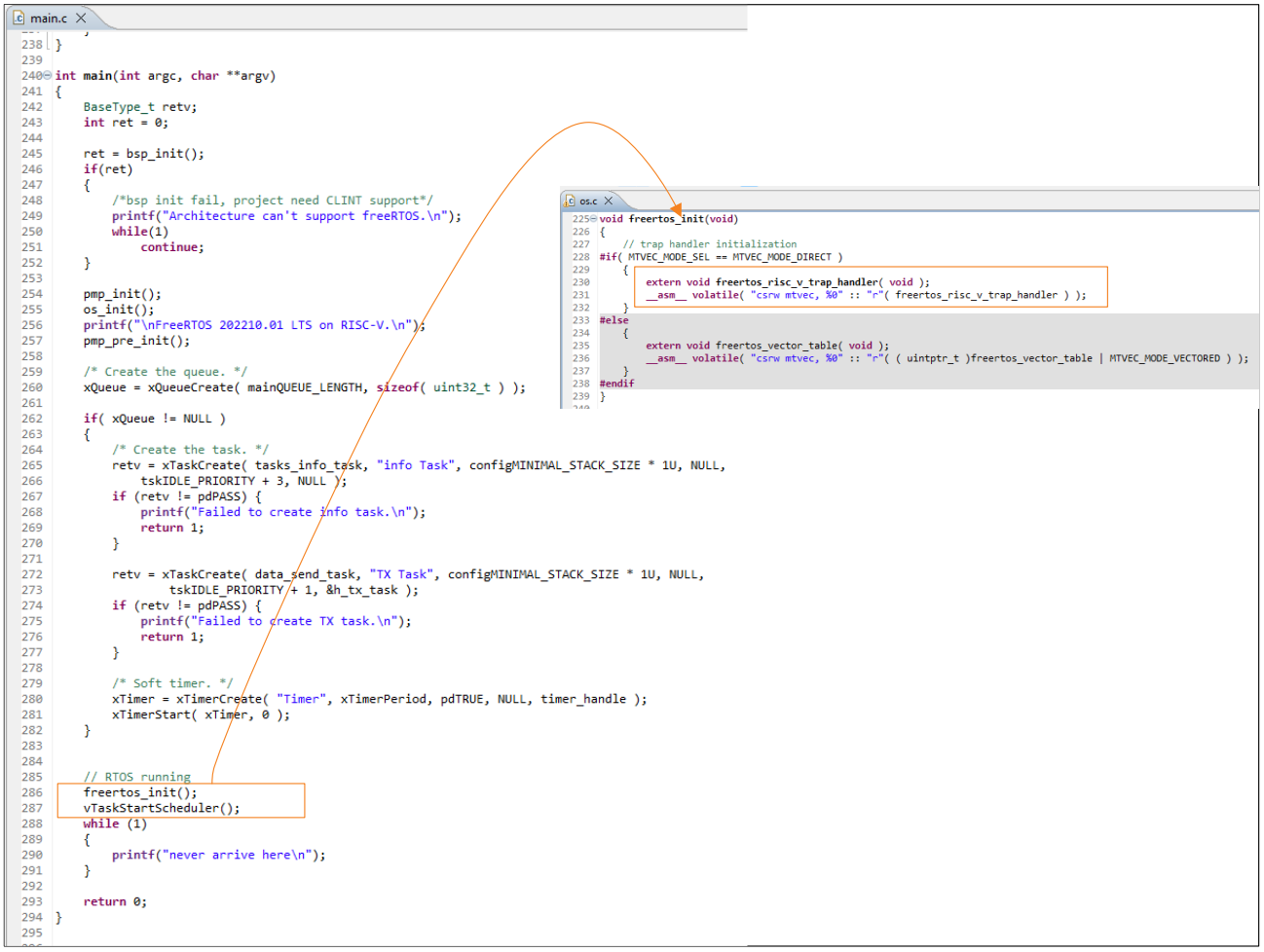

Figure 4.37. FreeRTOS Trap Handler Registration

The `freertos_risc_v_trap_handler` function is the entry point for handling all traps in FreeRTOS on RISC-V. The function begins by saving the current CPU context using `portasmSAVE_CONTEXT_INTERNAL`, which stores all necessary registers to the stack. This is crucial for preserving the execution state before handling the trap.
Next, the function reads the mcause and mepc registers to determine the cause of the trap and the program counter at the time the trap occurred. Based on the value of the mcause register, the handler branches into two paths:

- If the mcause register is non-negative (mcause[31] = b0), this value indicates a synchronous exception. The handler adjusts the mepc register to skip the faulting instruction and switch to the ISR stack. If the exception is an environment call (ECALL), a context switch is performed. Otherwise, control is passed to a user-defined exception handler. After handling, the context is restored and execution resumes.
- If the mcause register is negative (mcause[31] = b1), this value indicates an asynchronous interrupt. The handler saves additional context and calls handle_interrupt. If the interrupt is a machine timer interrupt, the system tick is updated and a context switch may be triggered. For other interrupts, control is passed to a user-defined interrupt handler. The context is then restored and execution resumes.

FreeRTOS trap entry point

Saves current CPU context (registers) to stack

Reads the trap cause and program counter at the time of the trap

If the mcause register is non-negative, branches to synchronous_exception.
If the mcause register is negative, proceed to asynchronous_interrupt.

Handles interrupts:

- Switches to ISR stack
- Checks if the interrupt is a machine timer interrupt
- If yes, updates timer, calls tick increment and context switch if needed
- Otherwise, jumps to user-defined application interrupt handler
- Before exit, restores the previously saved CPU context and resumes execution

Handles exceptions:

- Adjusts the mepc register to skip faulting instruction
- Switches to ISR stack
- If the exception is an environment call (ECALL), switch context
- Otherwise, jumps to user-defined application exception handler
- Before exit, restores the previously saved CPU context and resumes execution

Restores the previously saved CPU context and resumes execution

portASM.S

```
335 .section .text.freertos_risc_v_trap_handler
336 .align 8
337 freertos_risc_v_trap_handler:
338     portcontextSAVE_CONTEXT_INTERNAL
339
340     csrr a0, mcause
341     csrr a1, mepc
342
343     bge a0, x0, synchronous_exception
344
345 asynchronous_interrupt:
346     store_x a1, 0( sp )                 /* Asynchronous interrupt
347     load_x sp, xISRStackTop             /* Switch to ISR stack. *
348     j handle_interrupt
349
350 synchronous_exception:
351     addi a1, a1, 4                      /* Synchronous so update
352     store_x a1, 0( sp )                 /* Save updated exception
353     load_x sp, xISRStackTop             /* Switch to ISR stack. *
354     j handle_exception
355
356 handle_interrupt:
357 #if( portasmHAS_MTIME != 0 )
358
359     test_if_mtimer:                     /* If there is a CLINT th
360         addi t0, x0, 1
361         slli t0, t0, __riscv_xlen - 1   /* LSB is already set, sh
362         addi t1, t0, 7                  /* 0x8000[]0007 == machin
363         bne a0, t1, application_interrupt_handler
364
365         portUPDATE_MTIMER_COMPARE_REGISTER
366         csrr a0, mcause
367         csrr a1, mepc
368         call interrupt_check
369         call xTaskIncrementTick
370         beqz a0, processed_source       /* Don't switch context i
371         call vTaskSwitchContext
372         j processed_source
373
374 #endif /* portasmHAS_MTIME */
375
376 application_interrupt_handler:
377     call freertos_risc_v_application_interrupt_handler
378     j processed_source
379
380 handle_exception:
381     /* a0 contains mcause. */
382     li t0, 11                                   /* 11 == environm
383     bne a0, t0, application_exception_handler   /* Not an M envir
384     call exception_check
385     call vTaskSwitchContext
386     j processed_source
387
388 application_exception_handler:
389     call freertos_risc_v_application_exception_handler
390     j processed_source                  /* No other exceptions ha
391
392 processed_source:
393     portcontextRESTORE_CONTEXT
```

Figure 4.38. freertos_risc_v_trap_handler Implementation

FreeRTOS on RISC-V allows you to register custom handlers for interrupts and exceptions, named `freertos_risc_v_application_interrupt_handler` and `freertos_risc_v_application_exception_handler`, respectively. These functions are declared as __WEAK in FreeRTOS, where you can override the functions by providing your own implementations. When defined, these handlers are integrated into the trap handling flow and invoked based on the cause of the trap.
As shown in Figure 4.39, in Lattice FreeRTOS, custom handlers are implemented to manage interrupts and exceptions as follows:

- For interrupts, the trap handler calls `freertos_risc_v_application_interrupt_handler`, which is defined in the *os.c* file. This function receives the mcause and mepc registers values and then calls `isr_irqcall()`. The `isr_irqcall()` function is where you can implement your own logic to handle different interrupt sources. In the

example shown in Figure 4.39, the `isr_irqcall()` further calls `isr_callback()`, which uses a switch statement on the mcause register to dispatch the appropriate interrupt handler. This modular design allows you to define specific behavior for each interrupt type.

- For exceptions, the trap handler calls `freertos_risc_v_application_exception_handler`, also defined in *os.c* file. This function similarly receives mcause and mepc registers, and invokes `exception_check()`. Inside `exception_check()`, a switch statement is used to determine the type of exception and execute the corresponding handling logic. This allows the application to respond appropriately to exceptions such as illegal instructions, environment calls, or memory access faults.

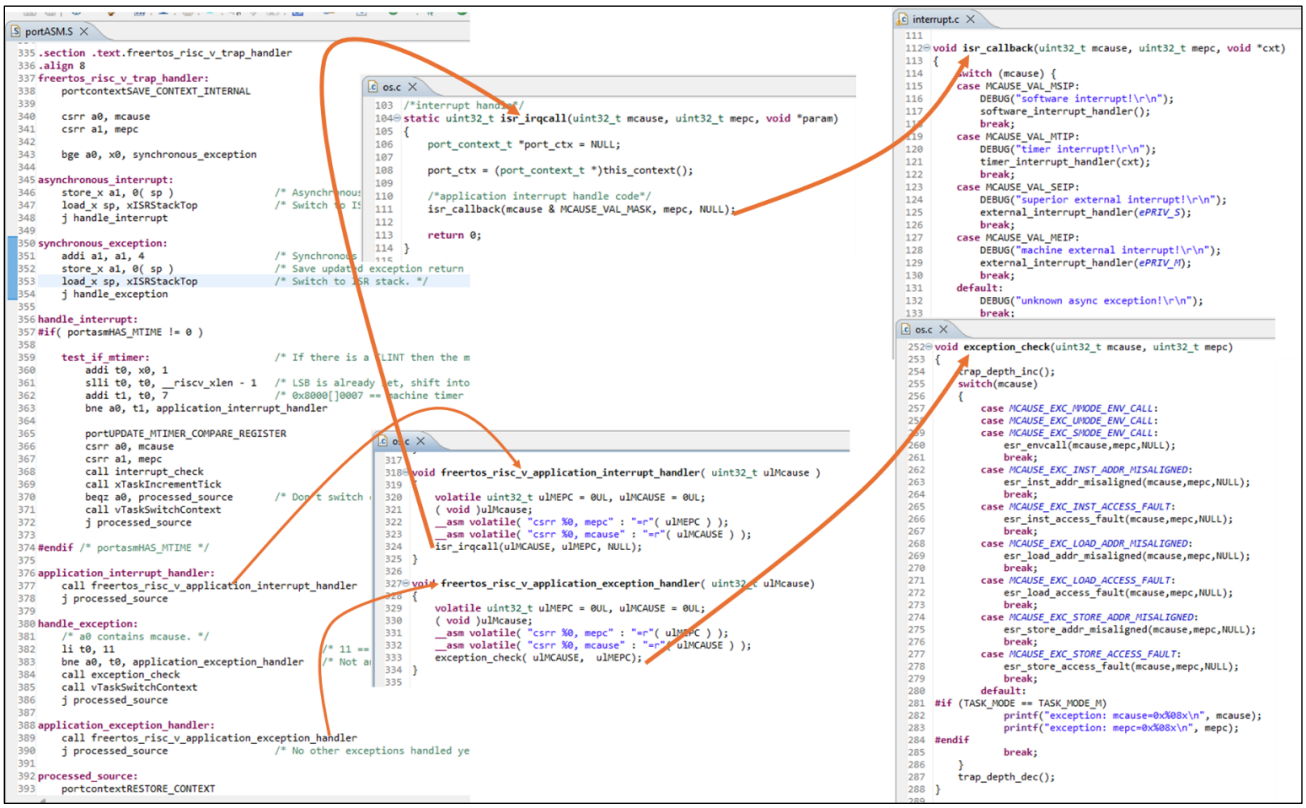

Figure 4.39. Interrupts Handling and Exceptions Handling in Lattice FreeRTOS

4.6.4.3. Vectored Mode Trap Handler

RISC-V RX advanced mode supports vectored mode interrupts in Bare Metal. To enable the vectored mode, you need to configure the mtvec register (Machine Trap-Vector Base Address Register) as follows:

- Set the `mtvec.MODE` bit to 1 to activate vectored mode.
- Set the `mtvec.BASE` field to the base address of the interrupt vector table.

When configured, asynchronous interrupts are dispatched to individual vector offsets using the formula:

BASE + 4 × interrupt_cause

Meanwhile, synchronous exceptions always enter at the base address (BASE), corresponding to offset 0x00.

In *riscv.h* file, you can define the MTVEC_MODE_SEL macro to MTVEC_MODE_VECTORED to enable vectored mode interrupt. This sets `mtvec.BASE` to trap_vector address (defined in *enry.S* file) and `mtvec.MODE` to 1.

```
#define MTVEC_MODE_SEL   MTVEC_MODE_VECTORED
```

Each offset in the vector table (trap_vector) is 4 bytes wide and contains a jump (`j`) instruction to the corresponding handler. This layout enables fast dispatch for specific interrupt causes.

RISC-V defines three primary machine-mode interrupt sources, each represented by a unique mcause register value. An interrupt is indicated when mcause.MSB = 1.

- Machine software interrupt (MSI, cause = 3): Used for inter-processor signaling or self-triggered events. For example, scheduling IPI.
- Machine timer interrupt (MTI, cause = 7): Generated by the core-local timer, commonly used for RTOS ticks and task preemption.
- Machine external interrupt (MEI, cause = 11): Originates from the platform-level interrupt controller (PLIC), aggregating peripheral interrupt lines.

These interrupt causes map directly to vector table entries at the following offsets:

- MSI (cause = 3) – 0x0C
- MTI (cause = 7) – 0x1C
- MEI (cause = 11) – 0x2C

The figure below shows a minimal example modification of a vectored trap handler tailored for machine exception and the three supported machine-mode interrupt sources. Only entries 3, 7, and 11 are specialized with fast-path handlers. All other entries fall back to a generic trap_entry.

```
# vectored interrupts and exceptions handlers.
.globl trap_vector
# the trap vector base address must always be aligned on a 4-byte boundary
.align 4
trap_vector:
    j       syncexc_entry        # BASE/Offset 0x00 Synchronous Exceptions
    j       trap_vector_entry1
    j       trap_vector_entry2
    j       msw_irq_entry        # Offset 0x0C Machine Software Interrupt (MSI)
    j       trap_vector_entry4
    j       trap_vector_entry5
    j       trap_vector_entry6
    j       mtime_irq_entry      # Offset 0x1C Machine Timer Interrupt (MTI)
    j       trap_vector_entry8
    j       trap_vector_entry9
    j       trap_vector_entry10
    j       mext_irq_entry       # Offset 0x2C Machine External Interrupt (MEI)
    j       trap_vector_entry12
    j       trap_vector_entry13
    j       trap_vector_entry14
    j       trap_vector_entry15

... existing code ...

syncexc_entry:
    # Handle synchronous exceptions
    ...
    mret

msw_irq_entry:
    # Handle MSI (cause = 3)
    ...
    mret

mtime_irq_entry:
    # Handle MTI (cause = 7)
    ...
    mret

mext_irq_entry:
    # Handle MEI (cause = 11)
      ...
    mret
```

Figure 4.40. Example Modification of a Vectored Trap Handler

4.6.5. Using the Lattice RISC-V BSP Interrupt Firmware

The differences between the programmable interrupt controller (PIC) of RISC-V MC and SM and the platform level interrupt controller (PLIC) of RISC-V RX include the driver firmware that is provided to initialize and control the hardware.

However, the BSP drivers for the PIC and PLIC utilize a table to support external interrupts. Each entry in these tables stores a pointer to the interrupt service routine function and a second pointer to context data for the interrupt service routine to process the interrupt.

Initialize these tables for the trap handling firmware in the BSP to call the appropriate interrupt service routine when an interrupt occurs. The PIC and PLIC drivers provide API calls to register external interrupts to the corresponding entries in the interrupt table. These calls also enable the interrupt in the interrupt controller hardware.

4.6.5.1. pic_isr_register()

The call signature for pic_isr_register() is:

```
unsigned char pic_isr_register(unsigned char src,
                               void (*isr) (void *),
                               void *context)
```

where,

src:	PIC input port number (interrupt number)
isr:	pointer to the ISR function
context:	void pointer to the associated context data structure

The `pic_isr_register()` function initializes the interrupt's entry in the `int_table` global array. The function also enables the interrupt in the PIC hardware.

The pointer to the context is passed as a void pointer to be generic. If you develop a driver for custom hardware, define a data structure to hold the information required by the ISR. Register pointers to instances of that data with BSP by casting the data as type void *.

The figure below shows the example on how to use `pic_int_register()` function to register an interrupt handler for a custom hardware driver.

```c
#include <stdint.h>
#include "pic.h"

// Define a context structure for the ISR
typedef struct {
    uint32_t device_id;
    volatile uint32_t *device_base;
} hw_context_t;

hw_context_t my_hw_ctx = {
    .device_id = 0,
    .device_base = (volatile uint32_t *)0x40000000  // Example base address
};

// Define the ISR function
void custom_hw_isr(void *context) {
    hw_context_t *ctx = (hw_context_t *)context;

    // Clear the interrupt flag (example logic)
    *(ctx->device_base + 1) = 0x1;

    // Handle the hardware event
    // (e.g., read status, update state, notify system, etc.)
}

// Function to setup custom HW interrupt. Called by main() function
void setup_custom_hw_interrupt() {
    uint8_t irq = 5;         // PIC IRQ number for the hardware

    int result = pic_int_register(irq, custom_hw_isr, (void *)&my_hw_ctx);

    if (result != 0) {
        // Handle error
    }
}
```

Figure 4.41. Example on Using `pic_int_register()` Function

4.6.5.2. plic_int_register()

The call signature for plic_int_register() is:

```c
int plic_int_register(uint8_t src,
                      uint8_t priority,
                      uint8_t mode,
                      irq_handler isr,
                      void *context)
```

where,

`src`:	PLIC input port number (interrupt number)
`priority`:	PLIC priority of the interrupt
`mode`:	privilege level of the interrupt, supervisor or machine level
`isr`:	pointer to the ISR function
`context`:	void pointer to the associated context data structure

The `plic_int_register()` function initializes the interrupt's entry in the plic_int_table. The function also initializes the PLIC hardware for the interrupt including the priority and enabling the interrupt at the correct privilege level.

The pointer to the context is passed as a void pointer to be generic. If you develop a driver for custom hardware, define a data structure to hold the information required by the ISR. Register pointers to instances of that data with BSP by casting the data as type void *.

The figure below shows the example on how to use `plic_int_register()` function to register an interrupt handler for a custom hardware driver.

```c
#include <stdint.h>
#include "plic.h"

// Define a context structure for the ISR
typedef struct {
    uint32_t device_id;
    volatile uint32_t *device_base;
} hw_context_t;

hw_context_t my_hw_ctx = {
    .device_id = 0,
    .device_base = (volatile uint32_t *)0x40000000  // Example base address
};

// Define the ISR function
void custom_hw_isr(void *context) {
    hw_context_t *ctx = (hw_context_t *)context;

    // Clear the interrupt flag (example logic)
    *(ctx->device_base + 1) = 0x1;

    // Handle the hardware event
    // (e.g., read status, update state, notify system, etc.)
}

// Function to setup custom HW interrupt. Called by main() function
void setup_custom_hw_interrupt() {

    uint8_t irq = 5;          // PLIC IRQ number for the hardware
    uint8_t priority = 2;     // Priority level
    uint8_t mode = 1;         // 1 = Machine mode, 2 = Supervisor mode

    int result = plic_int_register(irq, priority, mode, custom_hw_isr,
                                   (void *)&my_hw_ctx);

    if (result != 0) {
        // Handle error
    }
}
```

Figure 4.42. Example on Using `plic_int_register()` Function

5. RISC-V System Debugging

5.1. On-Chip Debug Support

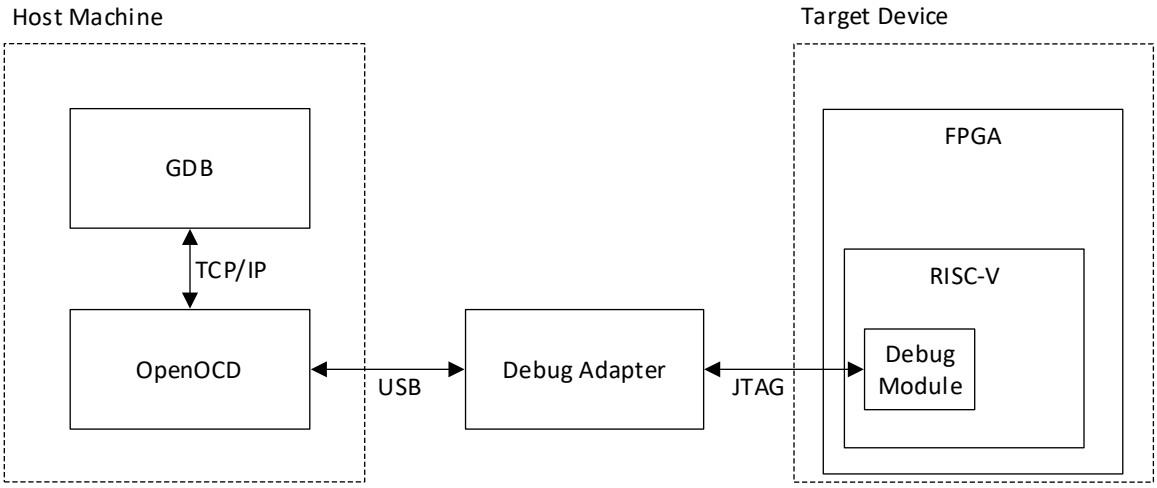

Figure 5.1. On-Chip Debug with GDB and OpenOCD

On-chip debug is essential in RISC-V embedded systems, enabling you to monitor and control software execution directly on hardware. This debug involves components like the debug module (DM) and support for breakpoints, watchpoints, and memory access. GNU debugger (GDB), a widely used debugger, connects to RISC-V targets via a debug server to inspect variables and manage execution flow. Open on-chip debugger (OpenOCD) acts as the debug server and bridges between GDB and the target device, translating commands into low-level operations that interact with the RISC-V debug interface, making this debug a key tool for development, testing, and system bring-up.

In the Lattice RISC-V architecture, on-chip debug functionality is available in the RX, MC, and SM variants, enabling you to utilize tools such as GDB and OpenOCD for software loading, boot-time configuration, and runtime debugging. However, the NANO variant does not support on-chip debug. As a result, you cannot rely on GDB or OpenOCD for interactive debugging or memory access during development.

Instead, the recommended approach for the NANO variant is to initialize software directly into memory using memory initialization files, such as .mem or .hex, during the synthesis or simulation phase. This method ensures the application is preloaded into the target memory, allowing the system to boot and operate without requiring external debug interfaces.

5.2. Hardware Debugging Using the JTAG Bridge IP

The JTAG Bridge IP enables efficient hardware debugging by providing direct access to memory and peripheral registers through the JTAG interface, bypassing the processor. This IP translates JTAG Test Access Port (TAP) signals into AXI4 transactions, supporting read and write operations. This capability is especially valuable for system bring-up and troubleshooting in embedded designs.

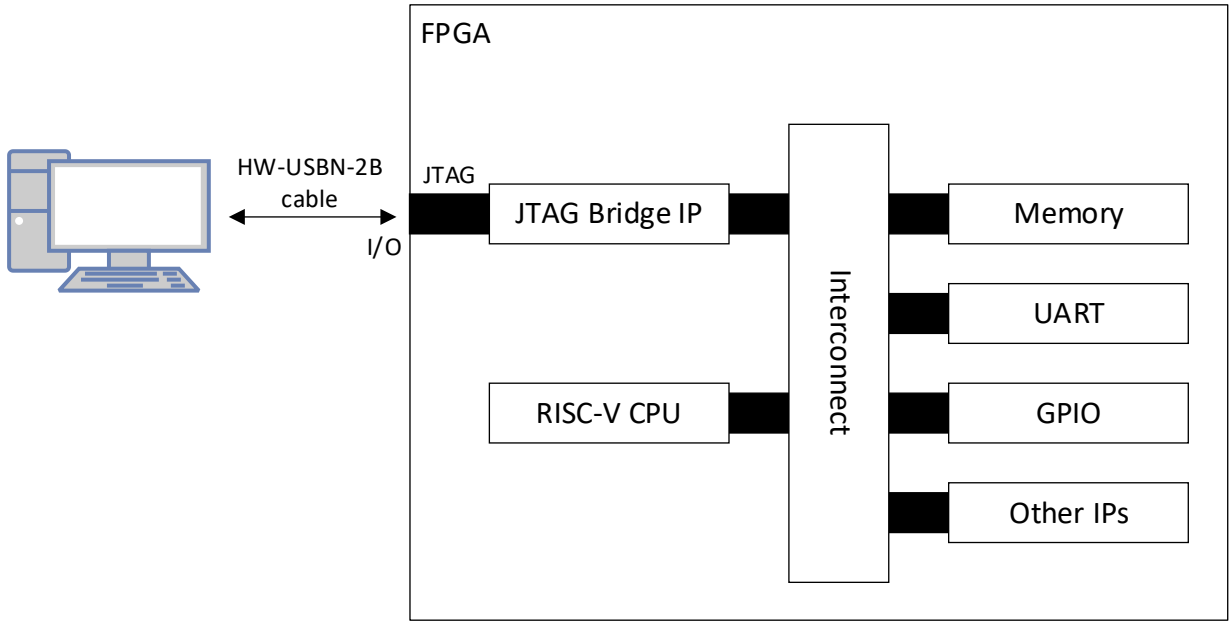

Figure 5.2. JTAG Bridge IP SoC Design

Key features supported by the JTAG Bridge IP are follows:

- AXI4 interface support
  Note: For AHB-Lite interconnect support, you may need to add the AXI4-to-AHB-Lite Bridge IP
- Configurable address width (7 to 32 bits)
- Data mask and size support

In a RISC-V SoC design, the JTAG Bridge IP enables access to memory-mapped registers for debugging purposes. You can use the provided TCL commands to perform read and write transactions by specifying the target address and data. The TCL commands allow writing data to or retrieving values from target address.

Integration of the JTAG Bridge IP requires the allocation of four additional FPGA pins to support the JTAG interface. To enable communication, connect these JTAG pins using the USB programming cable (HW-USBN-2B). Refer to the Programming Cables for PCs web page for details on the cable. The JTAG interface must be exported to the top level of the design and assigned to valid board pins. Additionally, the TCK signal must be mapped to the pclk pin to ensure proper functionality.

To use the JTAG Bridge IP for hardware debugging, follow these steps:

1. Include JTAG Bridge IP in the RISC-V SoC design.
   a. In the Lattice Propel Builder, select JTAG Bridge from IP Catalog.
   b. Configure the parameters as needed or use the default setting.
   c. Generate the IP instance.

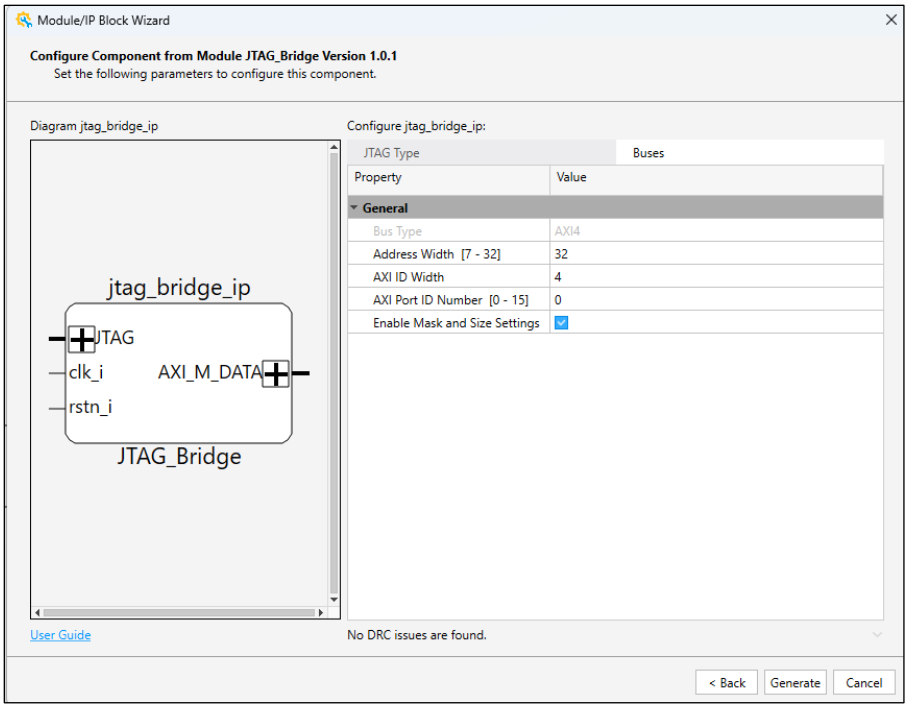

Figure 5.3. JTAG Bridge IP GUI Widget

2. Setup the JTAG Bridge interface signals connection.
   a. Export the JTAG interface signals to the top module through the Lattice Propel Builder GUI. Right click on the JTAG interface and click the **Export** button.
   b. Connect the AXI_M_DATA signal interface to interconnect.
   c. Connect rst_i and clk_i to reset source and clock source respectively.

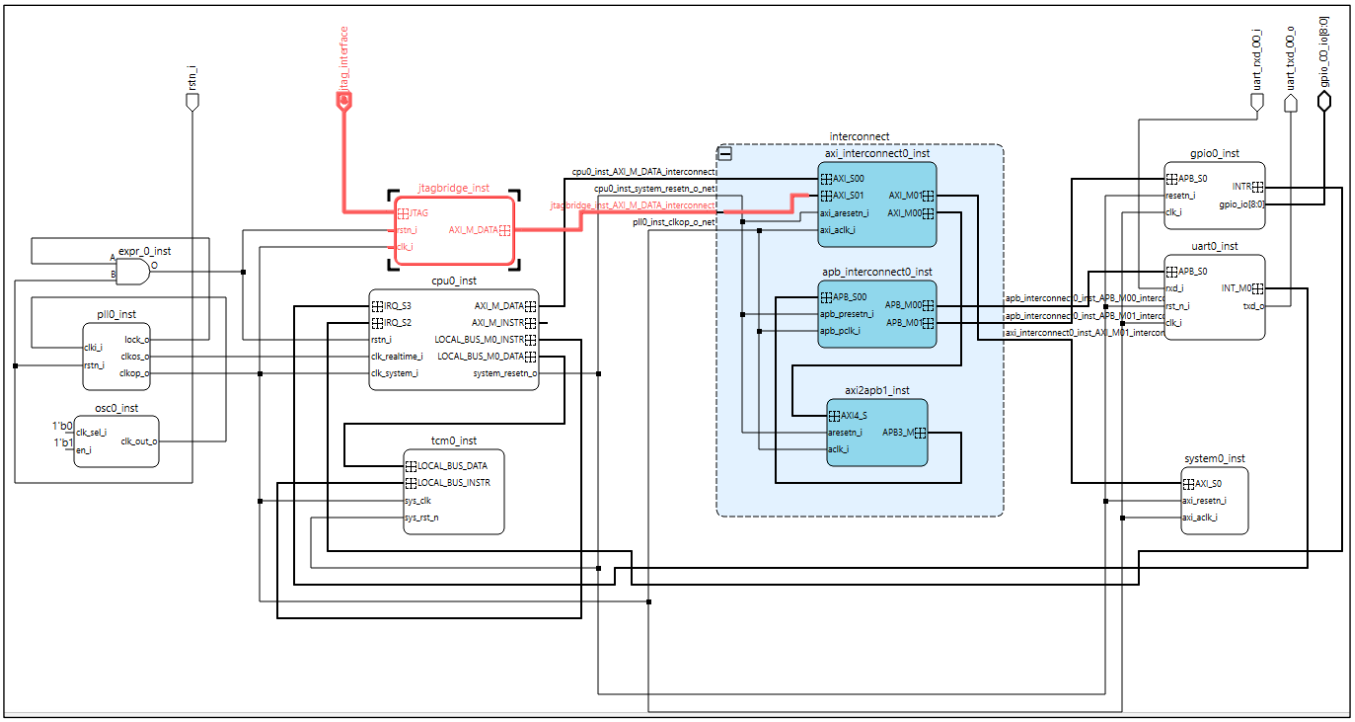

Figure 5.4. JTAG Bridge IP Connection

3. Assign JTAG ports to valid pins using the Device Constraints Editor or a post-synthesis constraint file in the Lattice Radiant software. Refer to the following examples of port constraint in the .pdc file for the Avant-E70 evaluation board.
   - ldc_set_location -site {V1} [get_ports jtag_interface_TDO_port]
   - ldc_set_location -site {W1} [get_ports jtag_interface_TDI_port]
   - ldc_set_location -site {W7} [get_ports jtag_interface_TMS_port]
   - ldc_set_location -site {T7} [get_ports jtag_interface_TCK_port]
4. Generate the bitstream and download the bitstream to the board.
5. Connect the HW-USBN-2B cable to the JTAG interface I/O ports. The table below shows the connection setup on the Avant-E70 evaluation board based on the port constraint configured in step 3.

Table 5.1. Lattice RISC-V Use Cases

HW-USBN-2B Cable	Avant-E70 Evaluation Board Connection
Red (Vcc)	J10 Header pin 1
White (TCK)	J10 Header pin 3
Orange (TDI)	J10 Header pin 7
Brown (TDO)	J10 Header pin 9
Purple (TMS)	J10 Header pin 13
Black (GND)	J10 Header pin 11

6. Debug via TCL console.
   a. Start the Lattice Propel Builder TCL Console.
   b. List the available USB port using the `sbp_cable list` command. Note that multiple ports may be listed. You need to identify the port that is pointing to the corresponding HW-USBN-2B cable.

```
% sbp_cable list
-------------------------------------------------------------------
 Port |    Name    |          Cable           | Channel | Location
-------------------------------------------------------------------
 0    | FTUSB-0    | Lattice HW-USBN-2B Ch A  | A       | 0
 1    | FTUSB-1    | Dual RS232-HS A          | A       | 2
-------------------------------------------------------------------
```

Figure 5.5. Available USB Ports

c. Open the desired port using the `sbp_cable open -port <port_number>` command.

```
% sbp_cable open -port 0
Open USB port 0 successfully.
```

Figure 5.6. Opening USB Port Pointing to the HW-USBN-2B Cable

d. Read data from target address of the SoC using the `sbp_read_memory -addr <address> -len <length>` command. The command returns data from the target address with the specified length.

```
% sbp_read_memory -addr 0x00200000 -len 8
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200000  00 00 00 00 22 22 11 11
%
% sbp_read_memory -addr 0x00200000 -len 16
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200000  00 00 00 00 22 22 11 11 44 44 33 33 66 66 55 55
%
% sbp_read_memory -addr 0x00200000 -len 32
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200000  00 00 00 00 22 22 11 11 44 44 33 33 66 66 55 55
0x00200010  88 88 77 77 AA AA 99 99 CC CC BB BB EE EE DD DD
%
% sbp_read_memory -addr 0x00200000 -len 64
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200000  00 00 00 00 22 22 11 11 44 44 33 33 66 66 55 55
0x00200010  88 88 77 77 AA AA 99 99 CC CC BB BB EE EE DD DD
0x00200020  FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
0x00200030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%
% sbp_read_memory -addr 0x00200000 -len 128
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200000  00 00 00 00 22 22 11 11 44 44 33 33 66 66 55 55
0x00200010  88 88 77 77 AA AA 99 99 CC CC BB BB EE EE DD DD
0x00200020  FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
0x00200030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00200040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00200050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00200060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00200070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%
```

Figure 5.7. Example of sbp_read_memory Command

e. Write data to the target address using the `sbp_write_memory -addr <address> -data <data>` command.

```
% sbp_write_memory -addr 0x00200030 -data 0x11
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200030  11
%
% sbp_write_memory -addr 0x00200030 -data 0x11223344
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200030  11 22 33 44
%
% sbp_write_memory -addr 0x00200030 -data 0x1122334455667788
-------------------------------------------------------------
Address     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
0x00200030  11 22 33 44 55 66 77 88
%
```

Figure 5.8. Example of sbp_write_memory Command

f. Close the port using the `sbp_cable close` command.

```
% sbp_cable close
Done to close USB port.
```

Figure 5.9. Closing the Port

5.3. Using OpenOCD Debugger and Reveal Analyzer

The Lattice Propel SDK supports software debugging using OpenOCD and GNU GDB to debug software related issues. For more information about the OpenOCD debugger, refer to the Lattice Propel SDK User Guide in the Lattice Propel Design Environment web page.

The Lattice Radiant software and the Lattice Diamond software provide the Reveal Analyzer for debugging Lattice FPGA designs. This debugging method monitors the FPGA logic and signals for finding hardware related issues. The Debugging with Reveal Usage Guidelines and Tips Application Note (FPGA-AN-02060) provides information about using the Reveal Analyzer.

Because problems encountered during FPGA embedded system development is difficult to identify whether it is software or hardware related, use the OpenOCD Debugger and Reveal tools to perform root cause investigation. An example is when the software code performs the correct sequences to setup the hardware but the expected behavior is not observed. Capture the hardware signals using the Reveal Analyzer to provide insight for the issues. Coupled with software debugger, software execution (using breakpoint) can be paused at the precise moment for the Reveal Analyzer to trigger on the desired conditions and capture the signals.

Figure 5.10 shows the general flow when debugging using the OpenOCD Debugger and the Reveal Analyzer.

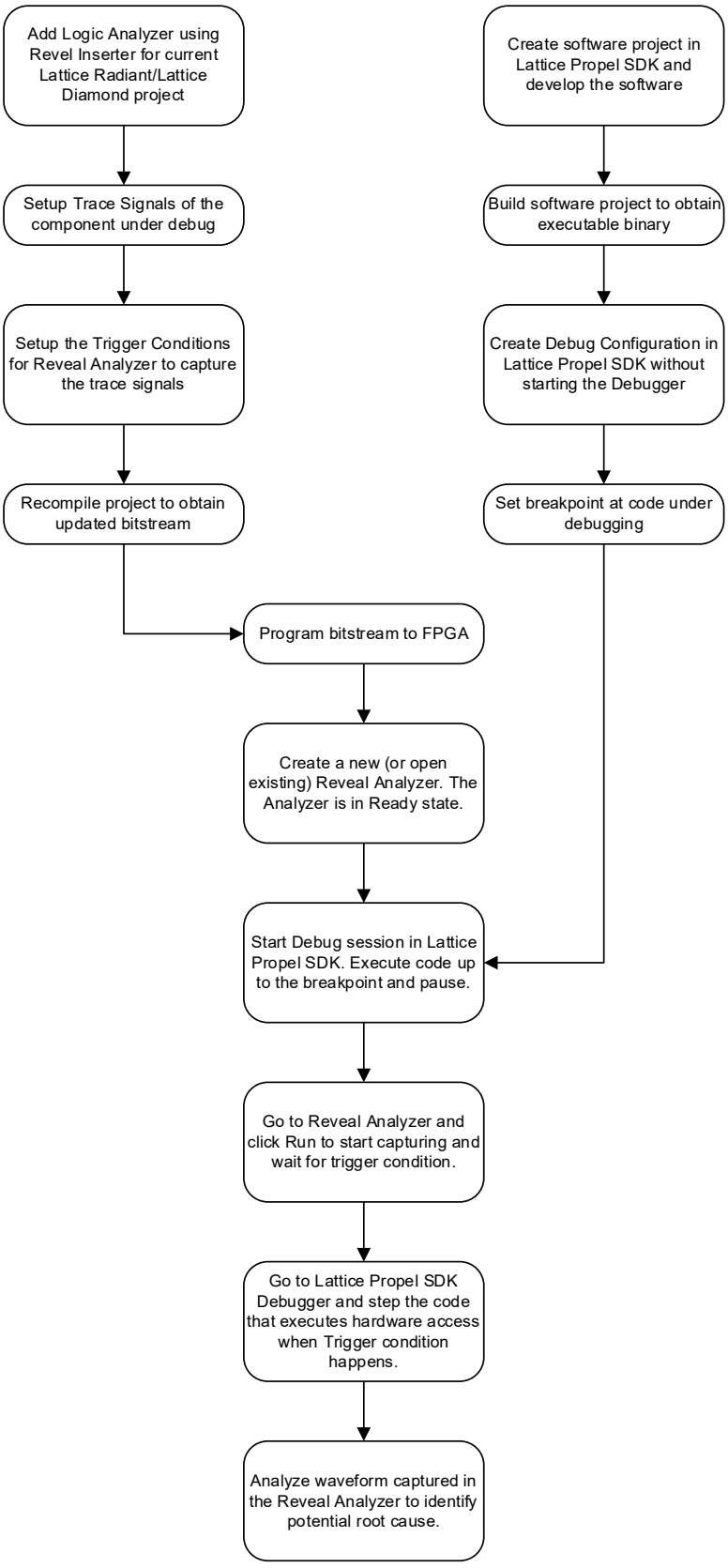

Figure 5.10. Debugging Flow Using the OpenOCD Debugger and the Reveal Analyzer

5.3.1. Known Limitations in Lattice Propel SDK

5.3.1.1. Incorrect Port Selection in Debug Configuration

Incorrect port selection in debug configuration within the Lattice Propel SDK results in the inability to detect the target device. To resolve this, ensure that the correct FTDI USB port is selected, corresponding to the JTAG programming channel. After selecting the correct port, scan for the device to verify that the board ID is detected successfully. The figure below shows the error when an incorrect port is selected.

```
Error: Location [0001] not found, please check your cable.
Error: Location [0001] not found, please check your cable.
```

Figure 5.11. Error for Incorrect Port Selection in Debug Configuration

5.3.1.2. JTAGHUB Crashed

The JTAGHUB may crash during operation, which typically causes the debugger to hang or lose connection. To recover from this, perform a power cycle on the device and ensure a stable power supply is available. When the system is stable, reprogram the device if necessary. The figure below shows the error when JTAGHUB crashed.

```
Error: JTAG scan chain interrogation failed: all zeroes
Error: Check JTAG interface, timings, target power, etc.
Error: Trying to use configured scan chain anyway...
Error: fpga_spinal.bridge: IR capture error; saw 0x00 not 0x01
```

Figure 5.12. Error When JTAGHUB Crashed

5.3.1.3. Incorrect JTAG Channel Settings

Incorrect JTAG channel settings can prevent the debugger from accessing the target CPU. It is important that the JTAG channel settings in the Lattice Propel SDK debug configuration match the JTAG channel selection defined in the RISC-V IP. You may also want to check if the JTAG pin constraints are configured correctly for the specific device to ensure proper signal routing and connectivity. The figure below shows the error when incorrect JTAG channel settings are configured.

```
Info : TAP fpga_spinal.bridge does not have IDCODE
Info : TAP auto0.tap does not have IDCODE
Info : TAP auto1.tap does not have IDCODE
Info : TAP auto2.tap does not have IDCODE
Info : TAP auto3.tap does not have IDCODE
Info : TAP auto4.tap does not have IDCODE
Info : TAP auto5.tap does not have IDCODE
Info : TAP auto6.tap does not have IDCODE
Info : TAP auto7.tap does not have IDCODE
Info : TAP auto8.tap does not have IDCODE
Info : TAP auto9.tap does not have IDCODE
Info : TAP auto10.tap does not have IDCODE
Info : TAP auto11.tap does not have IDCODE
Info : TAP auto12.tap does not have IDCODE
Info : TAP auto13.tap does not have IDCODE
Info : TAP auto14.tap does not have IDCODE
Info : TAP auto15.tap does not have IDCODE
```

Figure 5.13. Error for Incorrect JTAG Channel Settings

5.3.1.4. SoC Initialization Failure

SoC initialization failure is often caused by PLL not locking or reset signal assigned to the wrong pin. Check the PLL lock status and ensure the reset pin is constrained correctly for initialization. The figure below shows the error when SoC fails to initialize correctly.

```
Info : JTAG tap: fpga_spinal.bridge tap/device found: 0x10001fff (mfg: 0x7ff (<invalid>), part: 0x0001, ver: 0x1)
Info : Listening on port 3333 for gdb connections
Started by GNU MCU Eclipse
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : accepting 'gdb' connection on tcp/3333
Info : Halt timed out, wake up GDB.
Error: timed out while waiting for target halted
```

Figure 5.14. SoC Initialization Failure

5.4. Using Breakpoints

Breakpoints are a fundamental tool in the debugging process. They permit the designer to halt the program execution when reaching a prespecified condition. The designer can then perform typical debug functions such as checking the state of the CPU's registers and CSRs, verifying that variables in memory contain expected values, and observing program flow by single stepping the code following the breakpoint.

Lattice RISC-V implementations currently support two types of breakpoints: software breakpoints and hardware breakpoints. Both types of breakpoints halt program execution when reaching a particular location in the instruction memory. However, the approaches are different for the two types of breakpoints.

5.4.1. Hardware Breakpoints

A hardware breakpoint uses a comparator to test the value of the program counter against a value that debugger utility programs into the debug logic. When the PC matches the hardware breakpoint address, control of CPU execution is transferred to the debug core.

Execution resumes only when the debugger utility directs the CPU debug core to exit the debug state. The instruction that resides at the address of the hardware breakpoint is executed upon return from the debug state, not prior to entering it.

Lattice RISC-V CPUs support two hardware breakpoints.

5.4.2. Software Breakpoints

The RISC-V ISA defines a special instruction, EBREAK, that passes control to the debugging environment. The debugger uses the EBREAK instruction to implement software breakpoints.

When a developer sets a software breakpoint, the debugger utility identifies the address in program memory that corresponds to the targeted line of source code. The debugger utility replaces the instruction at that target address with the EBREAK instruction or the 16-bit equivalent, C.EBREAK, depending on the size of the original instruction.

When the program execution encounters the EBREAK instruction, control is transferred to the CPU debug module which halts the program execution. The developer can examine or modify the system state according to their debug strategy.

When the developer commands the debugger to resume program execution, the debugger performs the following operations:

1. Replace the EBREAK instruction with the original instruction.
2. Transfer control to the CPU to execute the original instruction.
3. Rewrite the EBREAK instruction back to the location of the software breakpoint.
4. Resume normal program execution.

The RISC-V CPUs support unlimited software breakpoints as software breakpoints do not rely on finite hardware resources.

However, when using software breakpoints, the debugger must be able to modify instruction memory at the address of the targeted instruction. If the debugger is unable to modify the instruction memory, the design can only use hardware

breakpoints. For example, a design that executes instructions from SPI flash memory via Execute in Place (XiP) feature must use hardware breakpoints as the debugger is unable to dynamically alter the SPI flash memory. Similarly, the Physical Memory Protection unit (PMP) for RISC-V RX could be configured to block write access to instruction memory following startup.

5.5. Using Semihosting

Semihosting is a mechanism that enables code running on the target to communicate with and use the I/O of the host computer. This method is useful during the development stage to output messages to the debug console without using the UART interface. Semihosting operates by stopping the processor execution and transferring the data from target to the host. Hence, semihosting generally does not provide high performance.

Lattice Propel SDK provides semihosting support. You can enable semihosting during project creation or changing the project property after project creation.

5.5.1. Enabling Semihosting During Project Creation in Lattice Propel SDK

Select Semihosting (--oslib=semihost) under System Library when creating a new C/C++ project in the Lattice Propel SDK as shown in Figure 5.15.

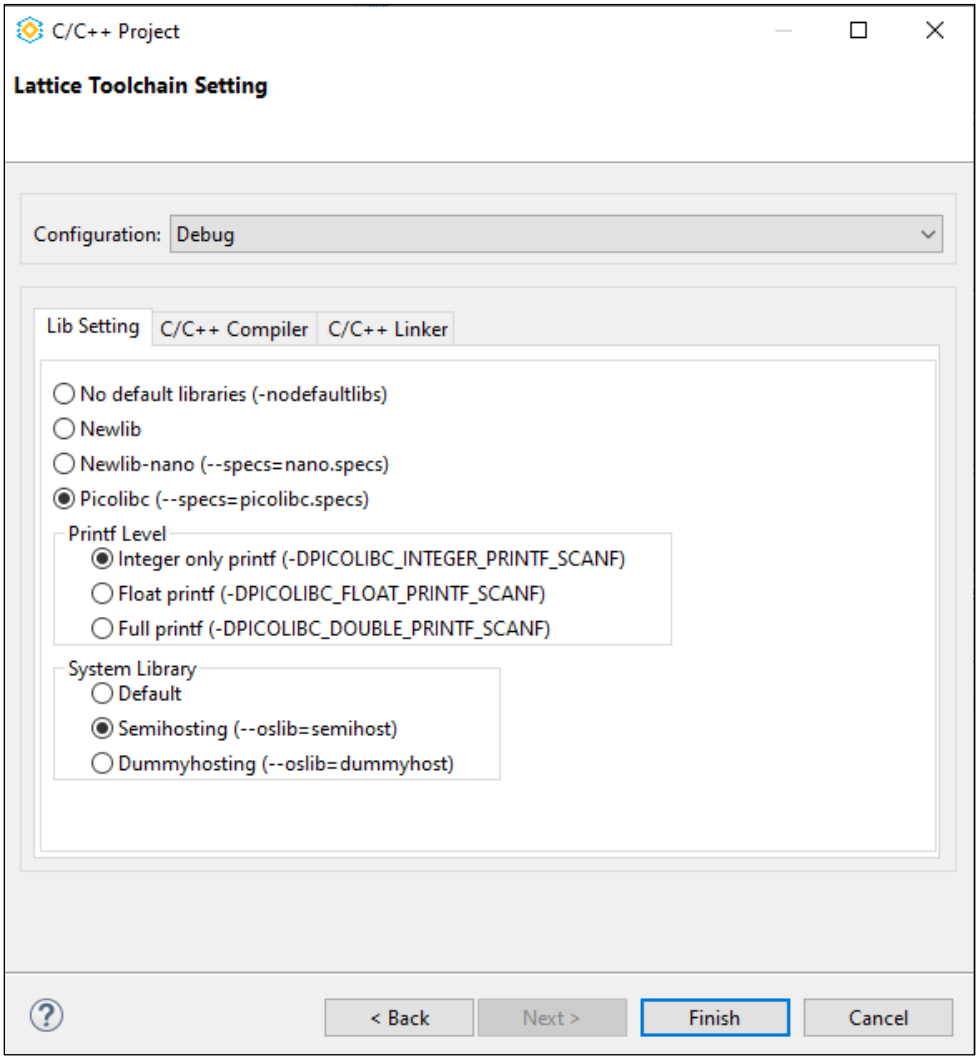

Figure 5.15. System Library Settings During C/C++ Project Creation

5.5.2. Enable Semihosting After Project Creation

If semihosting is not selected during project creation, you can change the project property to enable semihosting as follows:

1. Select the project in the Lattice Propel Project Explorer window.
2. Click **File > Properties**.
3. Select **C/C++ Build > Settings > GNU RISC-V Cross C Linker > Miscellaneous**.
4. In **Other linker flags**, type **--oslib=semihost**.
5. Click **Apply and Close**.
6. Click **Project > Build Project**.

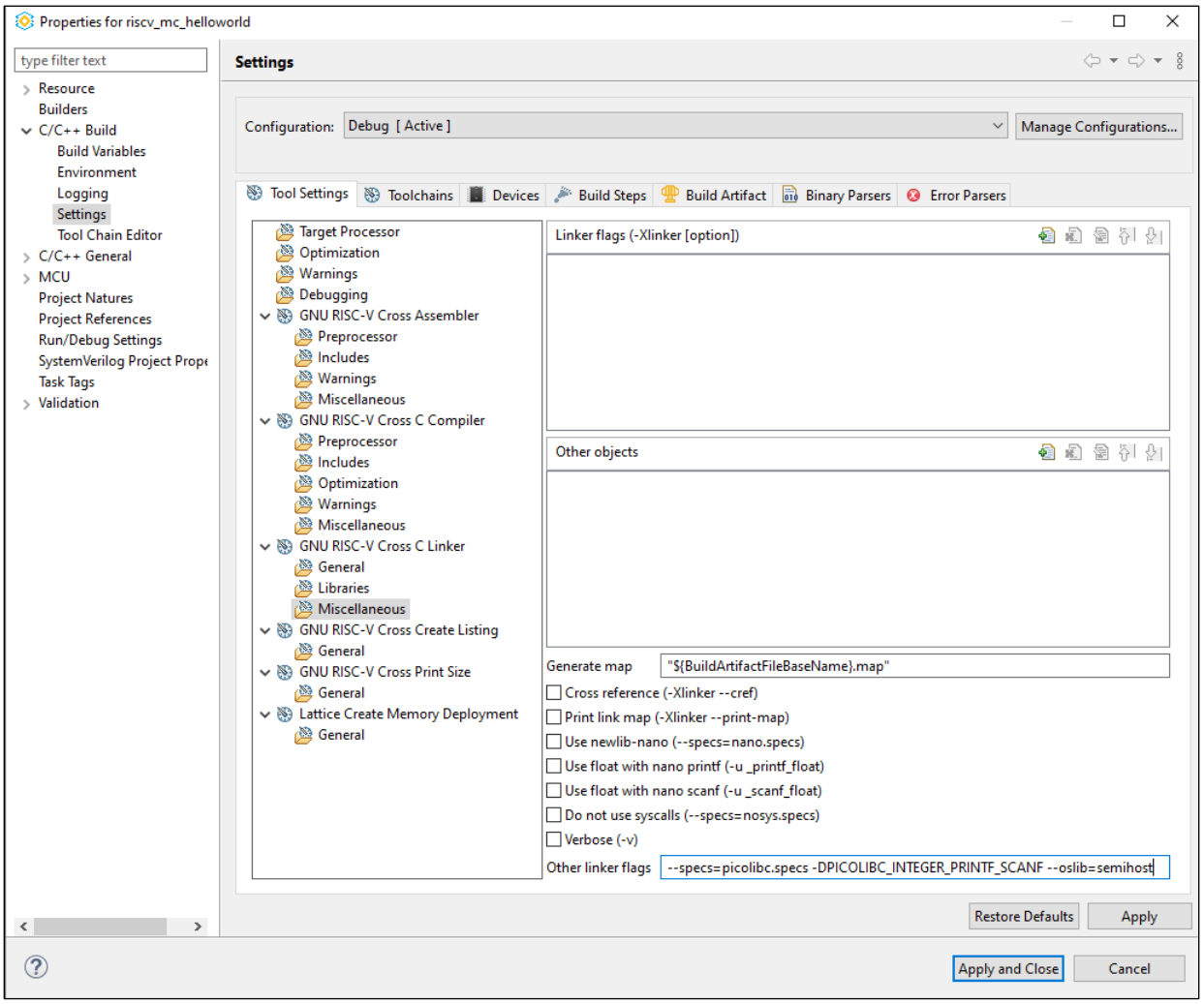

Figure 5.16. Enable Semihosting After Project Creation

Semihosting supports file I/O access to the host using fopen, fwrite, and fread functions. To enable this feature, set the heap size to non-zero in your project linker script as follows:

1. Expand the project in Project Explorer and locate the linker script in **src > linker.ld**.
2. Double click linker.ld.
3. Change the **HEAP_SIZE** value. You can set the value to 0x800 as shown in the example in Figure 5.17.

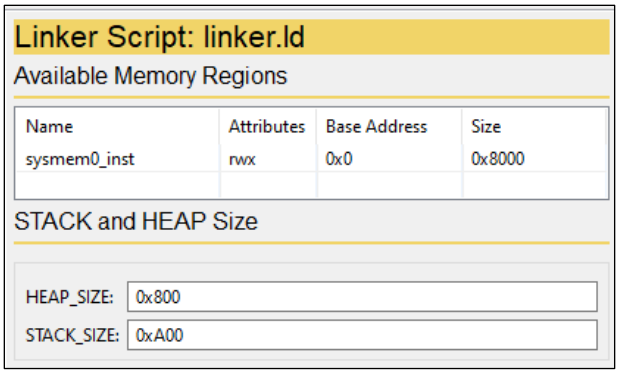

Figure 5.17. Changing Linker Script Heap Size

4. Write code in application that exercises fopen, fwrite, or fread.
5. Clean and rebuild the project.

5.6. Setting UART Serial Interface

Serial interface such as UART is a debug technique for printing messages to the console.

Most Lattice development boards have a component that converts the USB interface to UART. This conversion allows the board to send or receive UART signals to computer over the USB.

Figure 5.18 shows the Certus-Pro NX Evaluation Board which has the FTDI2232H chip for USB to UART interface. The UART signals are connected to the BDBUS channel of the chip, and shares the channel with the I2C interface. Install jumpers on the board to connect the UART signals to the FTDI2232H chip.

When assigning FPGA pins to the UART controller in your Lattice Propel Builder system, assign the UART IP TXD signal to the RXD pin and IP RXD signal to the TXD pin of the FTDI2232H chip.

When opening a Terminal program on computer (for example Putty or TeraTerm), the program may show 2 COM ports instead of 1. As there are 2 channels on the FTDI2232H chip, select the COM port that corresponds to the number of channels on the chip. The larger number COM port represents the UART interface.

Refer to the development board user guide and schematic when performing pin assignments and using the UART interface.

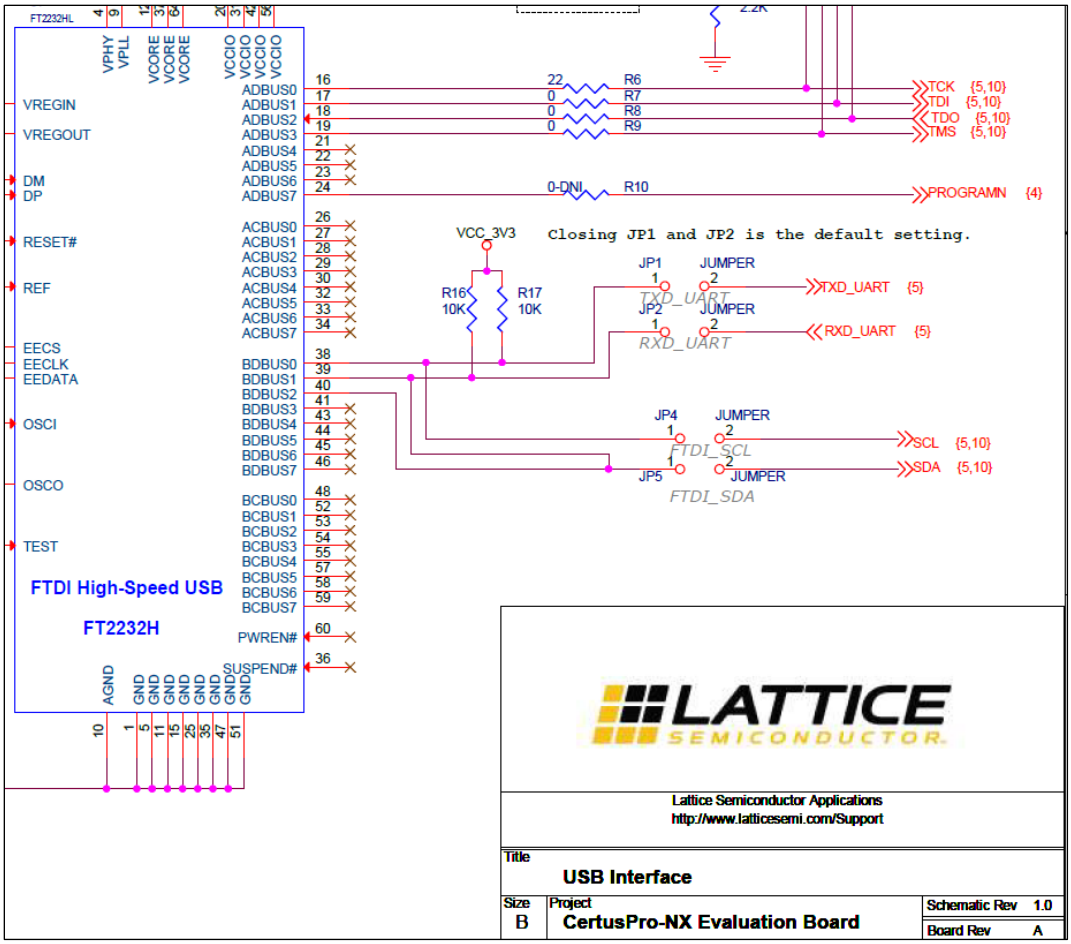

Figure 5.18. CertusPro-NX Evaluation Board UART Interface Schematic

References

- Lattice Propel 2025.2 Builder User Guide (FPGA-UG-02243)
- Lattice Propel 2025.2 SDK User Guide (FPGA-UG-02244)
- A Step By Step Approach to Lattice Propel
- RISC-V RX CPU IP – Lattice Propel Builder 2025.2 User Guide (FPGA-IPUG-02302)
- RISC-V MC CPU IP – Lattice Propel Builder 2025.2 User Guide (FPGA-IPUG-02300)
- RISC-V SM CPU (State Machine) IP Module User Guide – Lattice Propel Builder (FPGA-IPUG-02279)
- RISC-V NANO CPU IP – Lattice Propel Builder 2025.2 User Guide (FPGA-IPUG-02304)
- Mailbox IP User Guide (FPGA-IPUG-02306)
- Mutex IP User Guide (FPGA-IPUG-02307)
- RISC-V AXI4 I/O Physical Memory Protection IP User Guide (FPGA-IPUG-02283)
- RISC-V AHB-L IOPMP IP User Guide (FPGA-IPUG-02286)
- AHB-Lite to APB Bridge Module User Guide (FPGA-IPUG-02053)
- AHB-Lite to AXI4 Bridge IP User Guide (FPGA-IPUG-02242)
- CrossLink-NX web page
- Certus-NX web page
- CertusPro-NX web page
- MachXO2 web page
- MachXO3 web page
- MachXO3D web page
- MachXO5-NX web page
- Avant-E web page
- Avant-G web page
- Avant-X web page
- Lattice Nexus Platform web page
- Programming Cables for PCs web page
- RISC-V RX and LPDDR4 Memory Controller web page
- FreeRTOS web page
- GCC, the GNU Compiler Collection web page
- RISC-V Platform-Level Interrupt Controller Specification web page
- Lattice Propel Design Environment web page
- Lattice Radiant Software web page
- Lattice Diamond Software web page
- Lattice Insights 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/Support/AnswerDatabase.

Revision History

Revision 1.1, February 2026

Section	Change Summary
All	• Performed minor formatting and editorial edits. • Updated relevant terminology in accordance with inclusive language guidelines.
Acronyms in This Document	Updated list of acronyms.
Introduction	Updated the Overview section.

Lattice RISC-V Processors Family	• Added support for RISC-V NANO in the Lattice RISC-V Processors Family section.
	• Updated the following tables:
	  • Table 2.1. Lattice RISC-V Use Cases
	  • Table 2.2. Features Comparison of Lattice RISC-V Variants

RISC-V Embedded Hardware Design Guidelines	• Added support for RISC-V NANO and updated the following tables:
	  • Table 3.1. RISC-V Processor Reset Vector
	  • Table 3.2. Types of Memory Device
	  • Table 3.3. RISC-V Processor Caches
	  • Table 3.4. System Memory IP Features and Use Cases
	• Updated the address of the TCM in the Tightly-Coupled Memory (TCM) section.
	• Renamed section from *External SDRAM Memory* to External SDRAM, and updated section.
	• Updated the Flash Memory section.
	• Updated the Interconnects and Bridges section.
	• Added the Unified Interconnect section.
	• Updated the equation in the AXI ID section.
	• Updated Figure 3.5. AHB-Lite Interconnect IP Parameters – General Tab.
	• Added support for RISC-V NANO processor in the General Settings section.
	• Added peripherals connection and support for RISC-V NANO in the Using APB Interconnect IP section.
	• Updated the bridges in the Lattice Propel Builder in the Using Bridges IP section, and updated the subsections as follows:
	  • Updated AXI4-to-AHB-Lite Bridge, AXI4-to-APB Bridge, and AHB-Lite-to-APB Bridge subsections.
	  • Added the AHB-Lite-to-AXI4 Bridge subsection.
	• Removed the *Using Feedthrough IP* section.
	• Added the following sections:
	  • Integrating External IP via Feedthrough IP
	  • Mailbox IP
	  • Mutex IP
	  • IOPMP IP
	  • Cache Coherency
	  • Burst Transaction Support
	• Updated the Assigning Address Map section.

RISC-V Embedded Software Design Guidelines	• Added support for RISC-V NANO in the Software Project Support section.
	• Added RISC-V NANO processor driver in Table 4.1. BSP Components and Hierarchy.
	• Updated Figure 4.1. Update System and BSP Wizard.
	• Updated the GUI Based Linker Script Configuration section.
	• Added the Memory Initialization File Generation section.
	• Renamed section from *Interrupts* to Interrupts and Exceptions, and updated section.
	• Updated the Lattice RISC-V Interrupt Controller Hardware section as follows:
	  • Added RISC-V NANO in Table 4.4. Lattice RISC-V External Interrupt Controllers by Model.
	  • Updated the Programmable Interrupt Controller (PIC) and Platform Level Interrupt Controller

Section	Change Summary
	(PLIC) subsections. • Added the Lightweight Interrupt Merge Controller (LWIMC) subsection. • Added the Lattice RISC-V Exceptions section. • Updated the Lattice RISC-V Trap Handlers section as follows: • Added support for RISC-V NANO in Lattice RISC-V Trap Handlers and Bare Metal Trap Handler sections. • Updated the Lattice FreeRTOS Trap Handler subsection. • Added the Vectored Mode Trap Handler subsection. • Added an example on using the function to register an interrupt handler for a custom hardware driver in the pic_isr_register() and plic_int_register() sections.
RISC-V System Debugging	• Added the following sections: • On-Chip Debug Support • Hardware Debugging Using the JTAG Bridge IP • Updated the reference to the Lattice Propel software user guide in the Using OpenOCD Debugger and Reveal Analyzer section and added subsections.
References	Updated references.

Revision 1.0, February 2024

Section	Change Summary
All	Production release.

www.latticesemi.com