sensAI GARD Application Reference Design User Guide

Reference Design

FPGA-RD-02332-1.0

January 2026

Disclaimers

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

Inclusive Language

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

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
AHB-L	Advanced High-Performance Bus – Lite
AI	Artificial Intelligence
APB	Advanced Peripheral Bus
API	Application Programming Interface
AXI4	Advanced eXtensible Interface Four
AXI BD	Advanced eXtensible Interface Buffer Descriptor interface
AXI Pri	Advanced eXtensible Interface Primary streaming interface where Ethernet data transfer takes place
AWB	Automatic White Balancing
BD	Buffer Descriptor
BLE	Bluetooth Low Energy
BSP	Board Support Package
CM5	Compute Module Five
CNN	Convolution Neural Network
CPU	Central Processing Unit
CSI-2	Camera Serial Interface Two
DD	Defect Detection
D-PHY	Digital Physical Layer (MIPI PHY layer)
DRAM	Dynamic Random Access Memory
DWC	Data Width Converter
EBR	Embedded Random Access Memory
ECC	Error-Correcting Code
FPGA	Field-Programmable Gate Array
FTDI	Future Technology Devices International
GARD	Golden AI Reference Design
GPIO	General Purpose Input/Output
HMI	Human-Machine Interface
HRAM	Hyper Random Access Memory
HUB	Host-accelerated Unified Bridge
HWiL	Hardware-in-the-Loop
I/O	Input/Output
I2C	Inter-Integrated Circuit
IoT	Internet of Things
IP	Intellectual Property
ISP	Image Signal Processing
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LMMI	Lattice Memory Mapped Interface
MCS	Memory Configuration Storage
MIPI	Mobile Industry Processor Interface
ML	Machine Learning
MOD	Multi-Object Detection
MTBF	Mean Time Between Failures
NMS	Non-Maximum Suppression
OS	Operating System
OSC	Oscillator

Abbreviation	Definition
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
PCS	Physical Coding Sublayer
PHY	Physical Layer
PLL	Phase-Locked Loop
POR	Power-On Reset
POST	Power-On Self Test
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
RFS	Root Filesystem
RISC-V	Reduced Instruction Set Computer Five
RISC-V RX CPU	Real-time operating system that runs on a processor using the RISC-V instruction
RoT	Root-of-Trust
RPi	Raspberry Pi
RTL	Register Transfer Level
Rx	Receiver
SDK	Software Development Kit
SoC	System-on-Chip
SOM	System-On-Module
S-OTA	Secure Over-The-Air
SPD	Scratch Pad
SPI	Serial Peripheral Interface
SRD	System Requirement Document
TCM	Tightly-Coupled Memory
Tx	Transmitter
UART	Universal Asynchronous Receiver/Transmitter
UID	Unique Identifier
USB	Universal Serial Bus
VE	Vision Engine
XIP	Execute-In-Place

1. Introduction

The Lattice CertusPro™-NX FPGA-based production-ready system-on-module (SOM) provides a compact, reliable, and scalable platform that integrates compute, memory, connectivity, and acceleration capabilities into a single module. These SOMs simplify deployment, reduce time-to-market, and ensure industrial-grade robustness for edge AI applications. By leveraging advanced processors, AI accelerators, and rich I/O interfaces, SOMs enable the development of intelligent systems that operate efficiently in resource-constrained and rugged environments.

Edge computing is essential for real-time data processing, low-latency decision-making, and AI-driven insights closer to the data source. Traditional cloud architecture faces bandwidth limitations, latency constraints, and security concerns, making edge solutions critical for industries such as smart manufacturing, autonomous systems, healthcare, and IoT.

Lattice Golden AI Reference Design (GARD) addresses these requirements by providing a small form-factor, purpose-built and well-tested hardware, reusable AI models (through retraining), along with a firmware and software stack that accelerates design, development, and deployment of edge AI applications using Lattice FPGAs and sensAI™ solution stack.

The SOM includes the Lattice MachXO3D™ device (LCMXO3D-9400-UWG69), a 9400-LUT FPGA with flash-based configuration memory and enhanced security features. On the SOM, this device manages power-up sequencing of the Lattice CertusPro-NX (LFCPNX-100) voltage supplies and provides the 24 MHz clock. The SOM PCB is designed to support features such as bitstream monitoring, GPIO expansion, and level shifting through the MachXO3D device.

1.1. GARD Overview

GARD offers a unified scalable design, which is a proven, ready-to-use, go-to platform that addresses approximately 60% of the base design for a typical edge AI solution, accelerating solution development and reducing design turnaround time. The user-defined space enables you to implement use-case-specific functionality.

More details about each subsystem are provided in subsequent sections.

1.2. Quick Facts

Table 1.1. GARD Summary

FPGA Devices	Targeted Board	LFCPNX-100 based SOM (LFCPNX-SOM-EVN)
	Targeted Device	LFCPNX-100-9ASG256I
	Supported User Interfaces (internal)	• Advanced eXtensible Interface 4 (AXI4) • Advanced Peripheral Bus (APB)
	Supported User Interfaces (external)	I2C, UART, PCIe, MIPI-Rx, MIPI-Tx, and GPIO
Design Tool Support	RTL Implementation	• Lattice Propel™ Software 2025.1.1 • Lattice Radiant™ Software 2025.1.1
	RTL Synthesis	Synopsys Synplify Pro®
	Firmware	• Bare-metal code • ISR handling for peripherals • Support for Sony IMX 219 camera
	Software Stack (HUB & Host Application)	Raspbian ARM64 OS with Raspberry Pi Desktop (Bookworm 12): • Linux kernel 6.12.25+rpt-rpi-2712 • GCC 12.2.0, Make 4.3 • Python 3.11.2 Libraries: • cJSON 1.7.18 • libgpiod 1.6.5 • libusb 1.0.29
	AI-Model	LATTE, LSCQuant, and sensAI Compiler

1.3. Features

Key features of GARD include:

- Completely reconfigurable ML pipeline powered by the LFCPNX-100-256BGA SOM.
- RISC-V Subsystem:
  - RISC-V RX CPU runs at 108 MHz with 128 kB tightly-coupled memory (TCM)
  - Peripherals support: QSPI, I2C (100KHz), UART (up to 921,600 baud), GPIO, register bank, and PCIe
- ML Subsystem:
  - Advance Convolutional Neural Network (CNN) IP runs at 108 MHz
- Memory Subsystem:
  - Dual HyperRAM (HRAM) controller
- Application Subsystem:
  - Support for MIPI D-PHY Tx (2 or 4 lanes) and MIPI D-PHY Rx (1, 2, or 4 lanes) with resolution up to 4KP30
  - Scalable image signal processing (ISP) with automatic white balance (AWB) support
- MachXO3D device (LCMXO3D-9400-UWG69) for security and monitoring
- GARD platform firmware with API to access and manage various subsystems
- Host-accelerated Unified Bridge (HUB), which is a software stack that communicates with GARD from an ARM-Linux host

1.4. Naming Conventions

1.4.1. Nomenclature

The following nomenclature is used in this document:

- Boot Up – The process of starting the RISC-V RX CPU, loading the bare-metal application software from external SPI flash into the HyperRAM and executing the application.
- Bootloader – Code that initializes and configures various peripherals and loads the bare-metal application software into HyperRAM.
- Bare-metal application software – Software that is loaded into HyperRAM and executed by the RISC-V RX CPU at the end of the boot-up process.
- SPI Flash – Non-volatile external memory that stores the bit-files, bare-metal application software, and the multi-boot memory configuration storage (MCS) bitstream.
- Machine learning (ML) inference – The process of using a trained machine learning model to make predictions and/or take decisions on new and unseen data.

1.4.2. Signal Names

Signal names that end with:

- *_n* are active low (asserted when value is logic 0).
- *_i* are input signals.
- *_o* are output signals.
- *_io* are bi-directional input/output signals.

1.5. Prerequisites

The following sections describe the software and hardware requirements for compiling the reference design and for running the demonstration application.

1.5.1. Software Tool Requirements

The following software packages are required for running the GARD demos and for development:

- Lattice software packages:
  - Lattice Propel 2025.1.1 Package:
    - Lattice Propel SDK

    - Lattice Propel Builder
  - Lattice Radiant 2025.1.1 Package:
    - IP Packager
    - Radiant Software
    - QuestaSim
    - Programmer
- Lattice sensAI Studio
- Raspberry Pi (RPi) Imager 1.9.6
- rpiboot (Windows installer)
- PuTTY
- RealVNC Viewer
- Git (with Git Bash)
- Microsoft Visual Studio Code (with extensions)

1.5.2. Hardware Requirements

The following components are required:

- CertusPro-NX SOM Module (LFCPNX-SOM-EVN):
  - CertusPro-NX Package (LFCPNX-100-9ASG256I)
  - MachXO3D Package (LCMXO3D-9400)
- CertusPro-NX Carrier Module (LF-GEN-CR-PCBA)
- Raspberry Pi Compute Module 5 (CM5)
- Micro-USB Type-A UART Cable:
  - Included in the CertusPro-NX SOM board kit
  - Used for programming the bitstream and firmware, and for ensuring proper terminal prints
- Raspberry Pi Camera Module 2 or Arducam 8 MP IMX219 Camera with 4-lane support
- 5V 25W Power Adapter (External wall mount, class II)

2. System Architecture Overview

A typical edge AI application involves the following processes:

- Data acquisition
- Pre-processing
- AI or ML inferencing
- Post-processing
- Communication with the host

The GARD system architecture is developed to achieve reusability, flexibility, and scalability for building multi-domain edge AI solutions.

The partitioned architecture ensures smooth data flow between various functions. The architecture isolates the high-bandwidth paths (for example, ML IP) from low-bandwidth paths (for example, peripherals), ensuring that bandwidth-hungry processing engines are not starved.

Figure 2.1 depicts an overview of the GARD architecture.

GARD comprises the following components:

- GARD Hardware – Includes three modules:
  - CertusPro-NX SOM Module
  - CertusPro-NX Carrier Module
  - Raspberry Pi Compute Module 5
- GARD RTL – Fully verified, modular, and reusable RTL that enables you to build custom edge AI applications. It consists of multiple subsystems:
  - ML Subsystem – Hardware accelerator performing AI inference tasks
  - RISC-V Subsystem – Handles system initialization, control, and data flow management
  - Application Subsystem – Vision processing frontend performing image capture, pre-processing, and MIPI passthrough
- GARD-Platform Firmware – Foundational firmware exposing APIs to abstract hardware access and resource management, reducing development time
- GARD-Application Firmware – Use-case-specific firmware developed using GARD, such as:
  - Multi-Object Detection (MOD)
  - Defect Detection (DD)
  - Human-Machine Interface (HMI)
- HUB – ARM-Linux based software library framework that abstracts hardware interfaces exposed by the GARD or FPGA using inbox and Lattice-custom drivers
- Host Application – ARM-Linux application linked with HUB library, performing GARD interactions (data transfers, sensor, and metadata streaming), and incorporating built-in visualizers such as EdgeHUB and EVE.

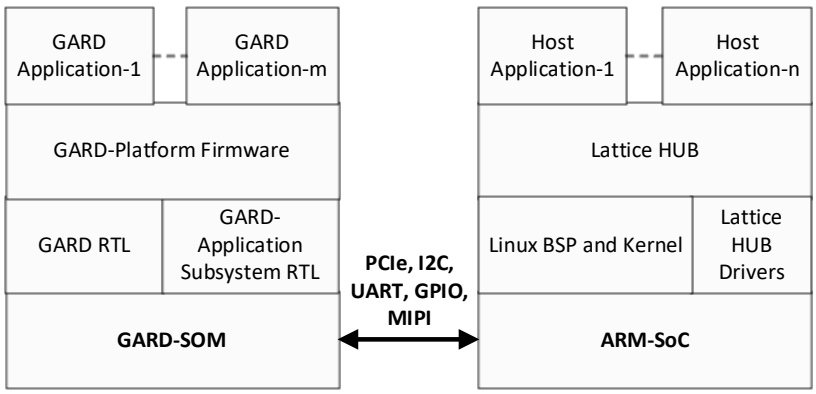

Figure 2.1. GARD Architecture Overview

2.1. General Data Flow

Refer to Figure 2.2, which depicts the data flow in a typical edge AI application using GARD.

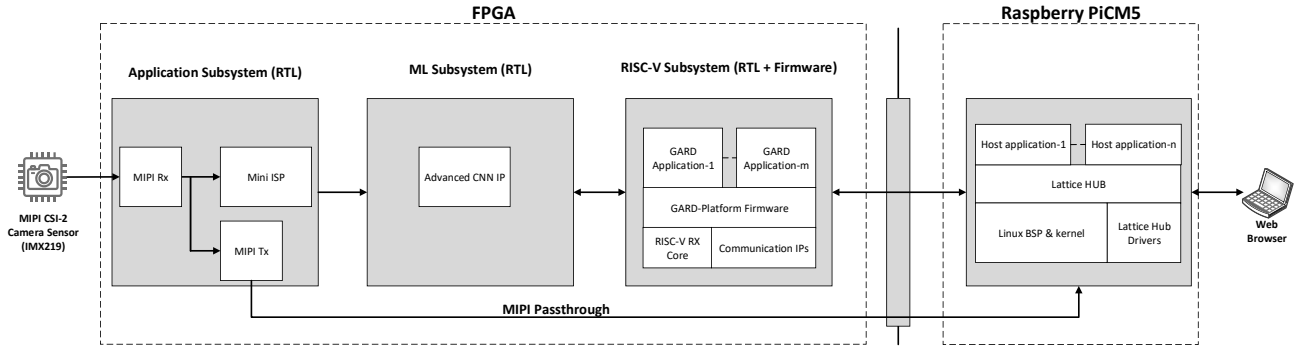

Figure 2.2. GARD Data Flow

The process starts with sensor data acquisition (in this case, a MIPI CSI-2 IMX219 camera). The Application Subsystem RTL (comprising video ingress and mini ISP) captures raw image data and processes the data before feeding it to the AI pipeline. The image pre-processing functions include:

- RAW10 to RGB conversion
- Scaling
- AWB
- Pixel normalization

A MIPI passthrough channel is also tapped from this subsystem for viewing the live feed.

The ML Subsystem RTL processes the pre-processed image using its AI pipeline and performs application-based inferencing. An advanced CNN ML engine is used for AI inferencing. The parallel processing feature for convolutional layers greatly reduces inference time.

The AI inference output from this block is fed to the RISC-V Subsystem (RTL + firmware). The firmware is layered into:

- GARD-Platform Firmware – Provides services such as system initialization, camera configuration, hardware-abstracted APIs for register access, and IP control or sequencing.
- GARD-Application Firmware – Performs post-processing (for example, non-maximum suppression (NMS) to filter overlapping bounding boxes and conversion of raw detection outputs into object labels and confidence scores). This output is referred to as AI metadata.

The AI metadata generated after post-processing is communicated to the host application running on the RPi host through the Lattice HUB framework.

On the Raspberry Pi CM5, Lattice HUB performs platform functions such as GARD Discovery, OS abstraction and data transfers over various bus interfaces. It also captures the live video feed from the passthrough using appropriate multimedia drivers and frameworks (V4L2).

The host application uses various rich-OS domain frameworks (such as OpenCV, libcamera, and V4L2) for processing and/or analytics, and software stacks (such as EdgeHUB and EVE Visualizer) for real-time rendering of AI metadata projected on live video feed input.

3. GARD Hardware

This section provides details on hardware boards used in the GARD system.

3.1. CertusPro-NX SOM Module

The key features of the CertusPro-NX SOM module include:

- CertusPro-NX FPGA (LFCPNX-100-ASG256)
  - 2-channel MIPI (4 lanes per channel)
  - 1-channel, 4 pairs SERDES
  - Ethernet RGMII interface
  - SPI Flash configuration
  - GPIO
- MachXOD3-4300 (LCMXO3D-9400-UWG69):
  - Hardware RoT (Root-of-Trust)
- Board resources:
  - 2 × 64 Mb HyperRAM
  - 2 × 100-pin connector for connection with the CertusPro-NX Carrier Module
- Programming circuit:
  - Programming software through USB/FTDI interface (JTAG)
  - Programming from onboard Flash (SPI)

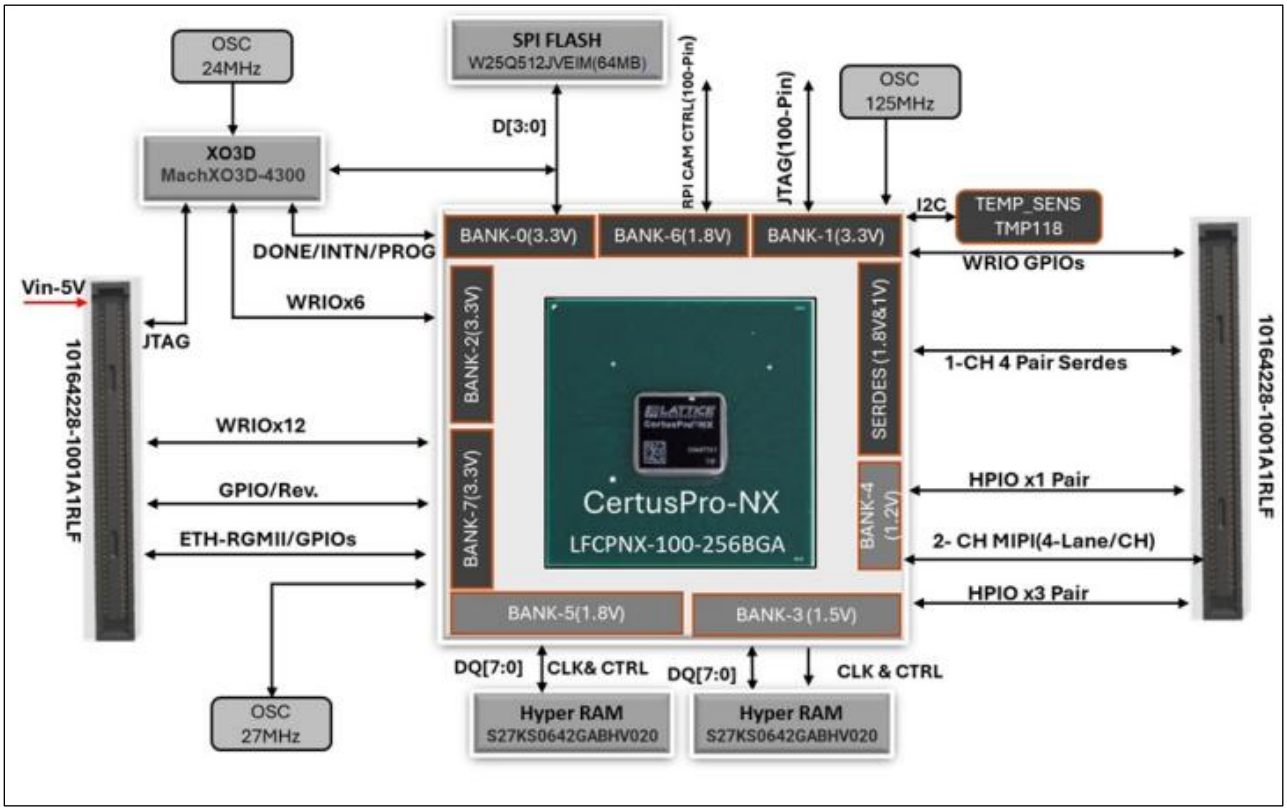

Figure 3.1. CertusPro-NX SOM Module Overview

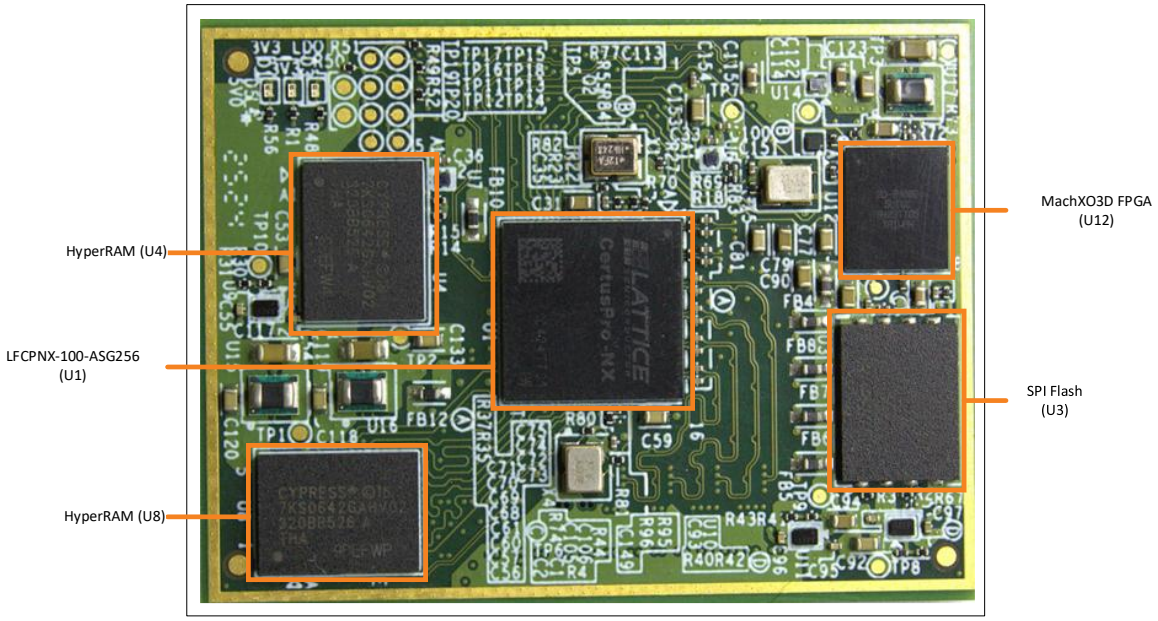

Figure 3.2. CertusPro-NX SOM Board

3.2. CertusPro-NX Carrier Module

The CertusPro-NX Carrier Module provides power management, storage, expansion and high-speed central interface hub between the SOM board and the Raspberry Pi CM5:

- Power and protection:
  - Power Connector (Type-C) – Accepts a 5 V input through a USB Type-C connector
  - Protection Circuit – Ensures safe delivery to the SOM board and Raspberry Pi CM5
  - Power Distribution – Supplies regulated 5 V to the SOM board and Raspberry Pi CM5
- Storage and USB:
  - MicroSD Card Slot – Provides storage for the Raspberry Pi CM5
  - USB 2.0 (DP, DN) – Differential pair for data transfer between the Raspberry Pi CM5 and external devices
  - USB 3.0 Type-A – Offers a high-speed external USB interface
  - u-USB Connector – Supports debugging or FTDI USB-to-serial communication
- Expansion connector – Extends GPIO signals from both the SOM and Raspberry Pi CM5 interfaces
- High-speed interfaces:
  - CSI-2 (4 data pairs + 1 clock pair) – Routes camera interface between the SOM board and Raspberry Pi CM5
  - CSI-2 (4 data pairs + 1 clock pair) – Dedicated MIPI CSI-2 camera connector for camera modules.
  - PCIe Gen2 x1 – Provides a high-speed expansion bus for the Raspberry Pi CM5 with control signals (CLK_REQ, WAKE#, PWR_EN, and PRSTn) for power and reset management
- Debug and communication:
  - FTDI USB-I/O – Converts USB to UART for debugging and programming
  - UART – Serial interface for the SOM board and Raspberry Pi CM5
  - JTAG-1 and JTAG-2 – Debugging and programming interfaces for the SOM board
- Networking
  - RJ45 Magjack – Provides a Gigabit Ethernet interface for the Raspberry Pi CM5

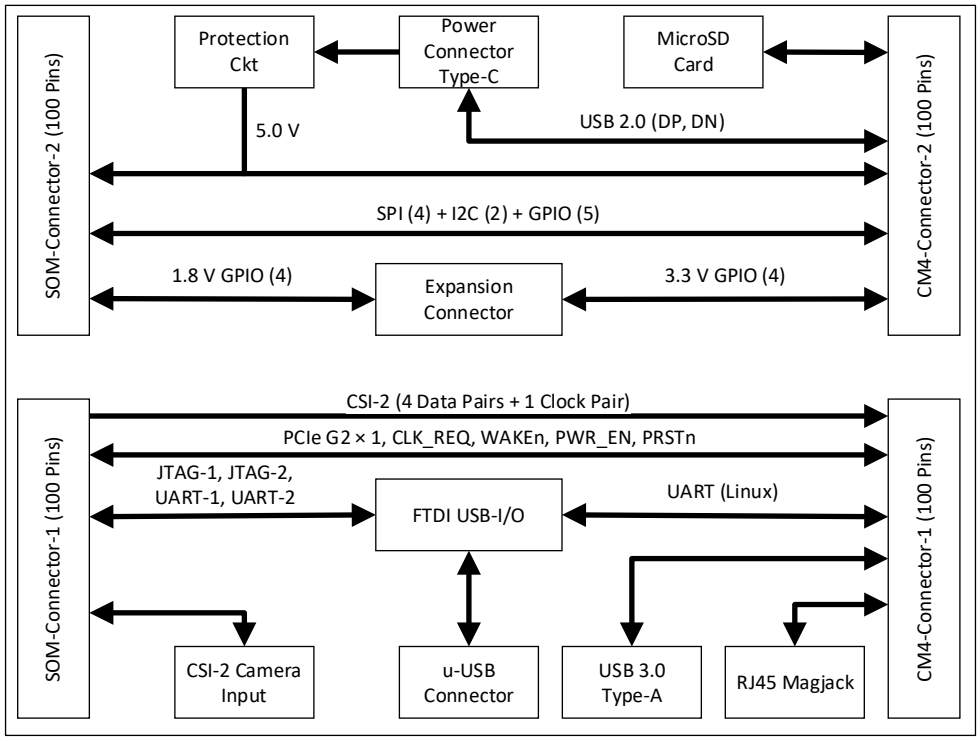

Figure 3.3. CertusPro-NX Carrier Module

3.3. Raspberry Pi Compute Module 5

Key features of the Raspberry Pi CM5 are as follows:

- Broadcom BCM2712, quad-core Cortex-A76 (ARMv8) 64-bit SoC at 2.4 GHz
- Small footprint: 55 mm × 40 mm × 4.7 mm module
- Four M2.5 mounting holes
- 4kp60 HEVC decoder
- OpenGL ES 3.1 graphics, Vulkan 1.2
- Options for 2 GB, 4 GB, or 8 GB LPDDR4-4267 SDRAM with ECC
- Option for certified radio module with:
  - 2.4 GHz, 5.0 GHz IEEE 802.11 b/g/n/ac wireless
  - Bluetooth 5.0, BLE
  - On-board electronic switch to select between PCB trace or external antenna
- Gigabit Ethernet PHY supporting IEEE 1588
- One PCIe 1-lane host, Gen 2 (5Gbps)
- One USB 2.0 port (high speed)
- Two USB 3.0 ports, supporting simultaneous 5 Gbps operation
- Up to 30 GPIO pins supporting either 1.8 V or 3.3 V signaling and peripheral options:
  - Up to five UART
  - Up to five I2C
  - Up to five SPI
  - One SDIO interface
  - One DPI (parallel RGB display)
  - One I2S
  - Up to four PWM channels
  - Up to three GPCLK outputs
- Two HDMI 2.0 ports (supports up to 4Kp60 on each port simultaneously)
- Two 4-lane MIPI ports supporting both DSI (display port) and CSI-2 (camera port)

- Single +5 V PSU input supports USB PD for up to 5 A at 5 V
- Real-time clock (RTC), powered from external battery

For more detail, refer to the Raspberry Pi CM5 documentation.

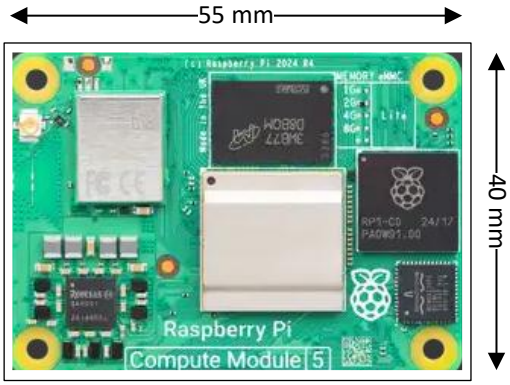

Figure 3.4. Raspberry Pi CM5

3.4. System Stack

The view of the entire system after stacking the CertusPro-NX SOM Module, CertusPro-NX Carrier Module, and Raspberry Pi CM5 is shown below.

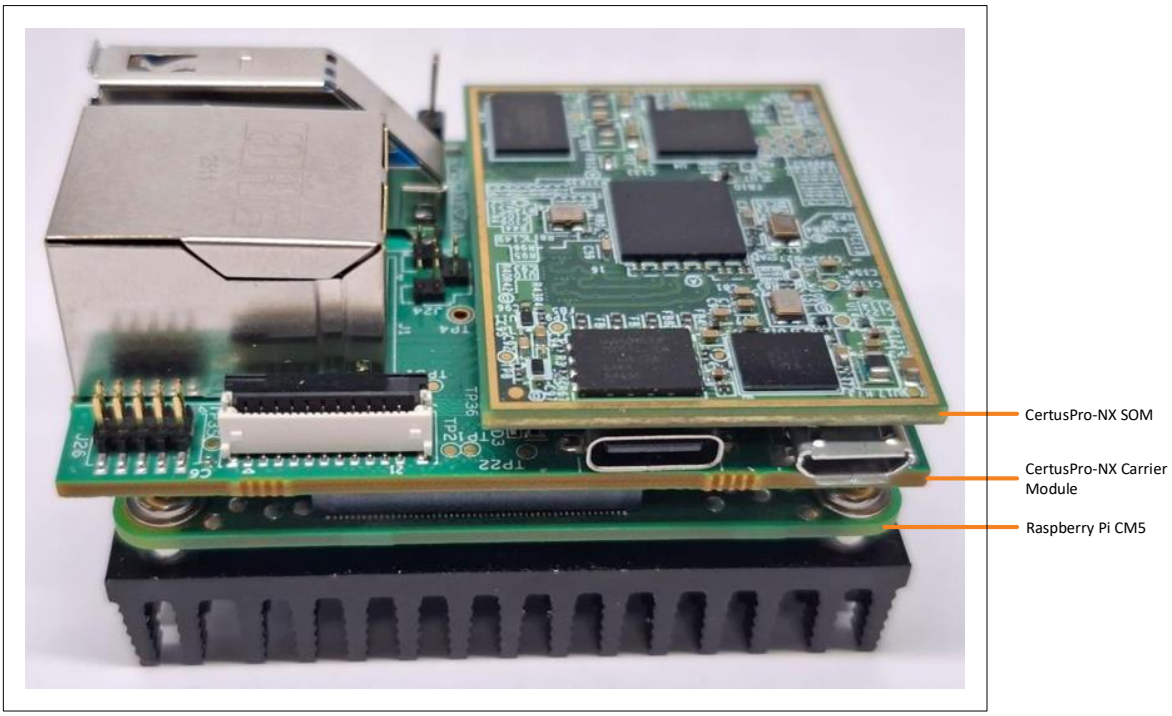

Figure 3.5. System Stack

4. GARD RTL

Figure 4.1 shows the GARD RTL, which is responsible for bringing up the entire system (in conjunction with associated peripherals) and performing the requisite AI/ML operations on data acquired from the sensors.

As shown in Figure 2.2, the GARD RTL is divided into:

- Application Subsystem – Responsible for data acquisition and pre-preprocessing.
- RISC-V Subsystem – Responsible for firmware execution and post-processing.
- ML Subsystem – Responsible for AI/ML inference.
- Memory and Host Subsystem – Responsible for memory communication and data exchange.

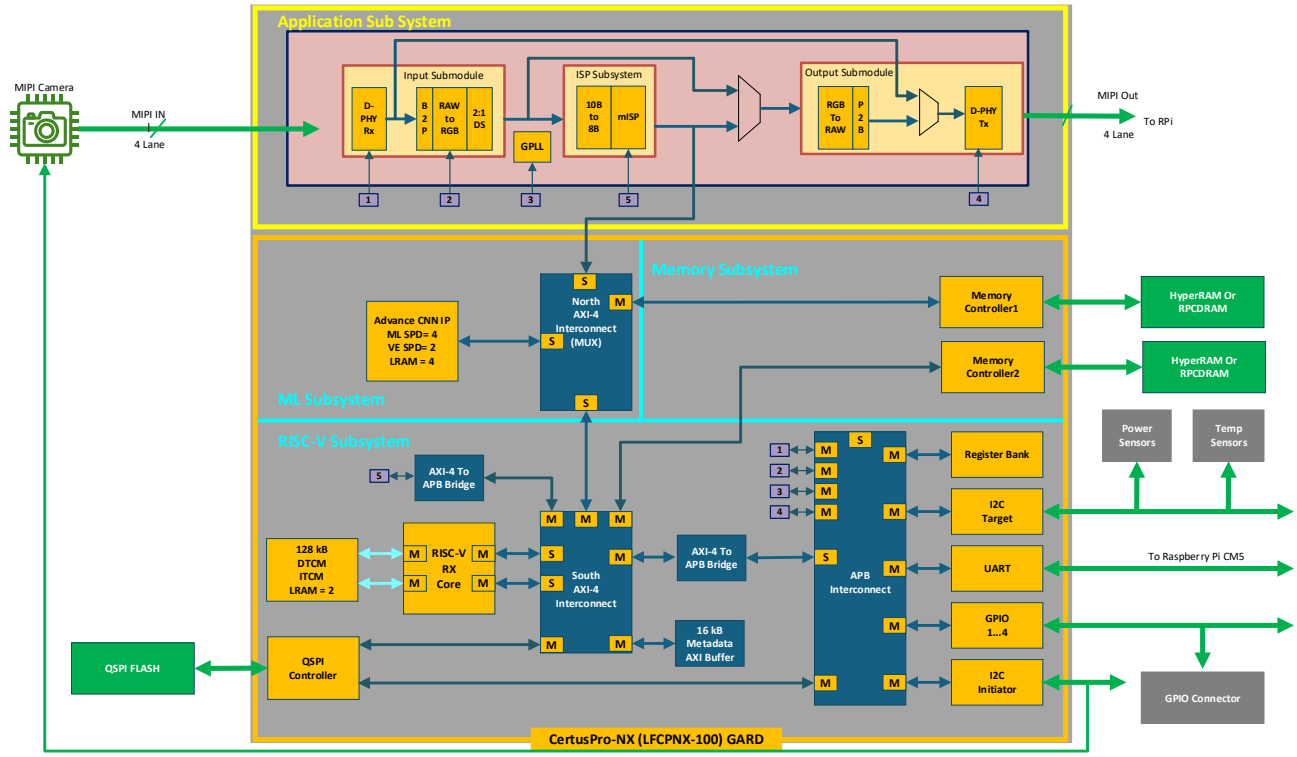

Figure 4.1. GARD RTL Overview

4.1. Clocking Scheme

The 24 MHz clock source available on board is used to generate all required clocks with the help of two PLLs. The GARD PLL generates clocks for the GARD RTL, while the Application PLL generates clock for logic that resides in the Application Subsystem. The figure and table below describe the GARD RTL clocking scheme.

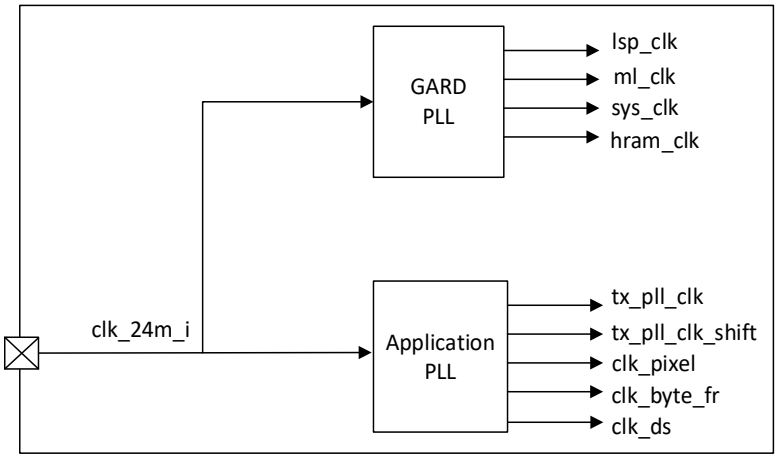

Figure 4.2. Clocking Scheme

Table 4.1. GARD Clocking Overview

Clock Name	Clock Frequency (MHz)	Clock Source	Destination
clk_24m_i	24	MachXO3D FPGA	PLL
lsp_clk	54	Internal PLL	UART, GPIOs, I2C target, I2C controller
ml_clk	120	Internal PLL	ML IP
sys_clk	108	Internal PLL	RISCV IP, OSPI controller, TCM, Interconnects, ML AXI
hram_clk	270	Internal PLL	HyperRAM controller
tx_pll_clk	360	Internal PLL	MIPI Tx D-PHY
tx_pll_clk_shift	360	Internal PLL	MIPI Tx D-PHY (90-degree shifted)
clk_pixel	144	Internal PLL	Video pipeline
clk_byte_fr	90	Internal PLL	Video pipeline
clk_ds	36	Internal PLL	Video pipeline

4.2. Reset Overview

The global reset (active low) to LFCPNX-100 is driven from the MachXO3D FPGA design. This incoming reset is fed to the RISC-V CPU, and the output system_resetn from the RISC-V is connected to all peripherals and IPs in the design. The ML IP has a separate software reset implemented in the GARD register bank. The figure and table below describe the reset overview and scheme.

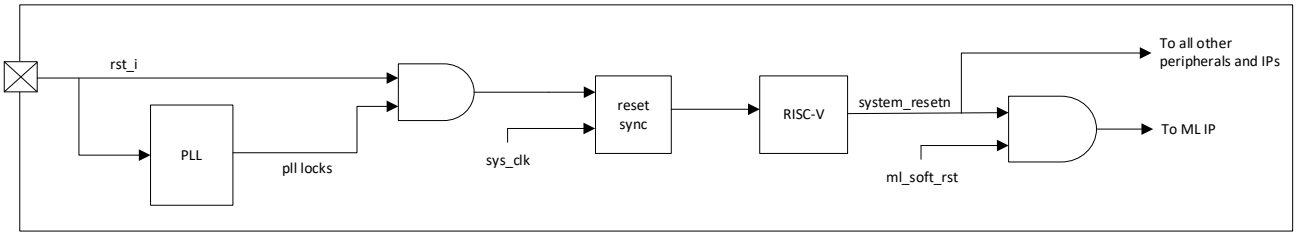

Figure 4.3. Reset Overview

Table 4.2. Reset Scheme

Reset Signal	Source	Destination	Description
rstn_i	MachXO3D FPGA	PLL reset input pin	Resets the PLL and initiates the system reset.
cpu_rstn_sync	Reset sync output port	RISC-V CPU reset input	CPU reset pin.

Reset Signal	Source	Destination	Description
system_resetn	RISC-V CPU output	All components in system	RISC-V CPU output reset pin, which provides reset to all components in the design. This signal also triggers a reset during CPU OCD debugging mode.
ml_rstn	ML software reset and with system_resetn	ML IP	Resets the ML IP.

4.3. Application Subsystem

The Application Subsystem is a user-defined space. Based on the end application, you define the video pipeline features and the pre- and post-processing blocks required to support them. You may also enable or disable ISP features based on the application. The required interfaces are provided in the GARD to connect with the application space.

The Application Subsystem is divided into three submodules: Input Submodule, ISP Submodule, and Output Submodule, as shown in the figure below.

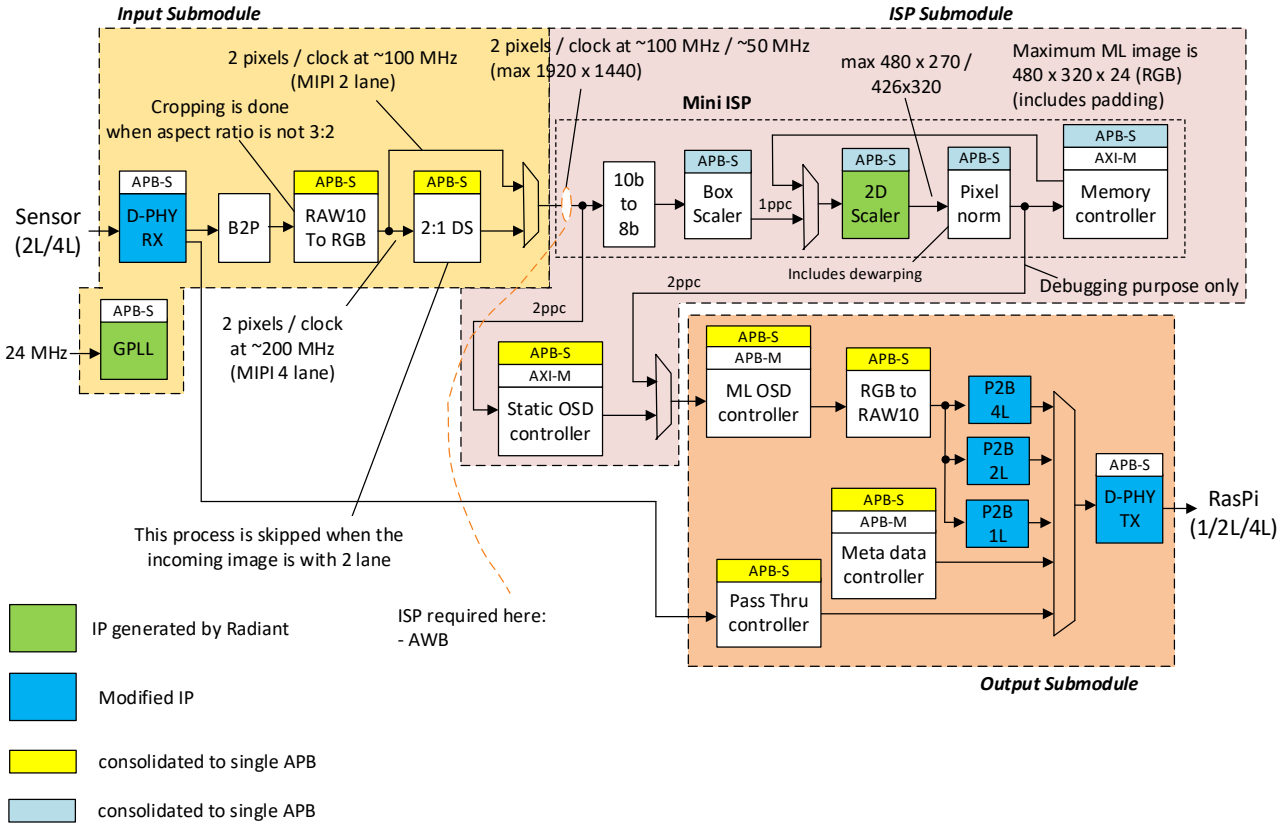

Figure 4.4. Application Sub System

4.3.1. MIPI In

MIPI CSI-2 is a standard interface for image data transfer from CMOS image sensors. GARD supports both 2-lane and 4-lane configurations. It supports a non-continuous MIPI clock and provides bandwidth of up to 1,500 Mbps per lane.

The maximum image resolution supported is 3,840 × 2,880 pixels. A 4-lane configuration is required for resolutions larger than 1,080p. Frame rates of up to 30 fps are supported. The system is tested using the native resolution of the IMX219 sensor, which is 3264(H) × 2464(W).

4.3.2. RTL Compile Option

During design compilation, you can minimize the design by enabling or disabling the following parameters.

Table 4.3. RTL Compile Options for Application Subsystem

Parameter	Values (Default in Bold)	Description
MIPI_RX_4L_FIXED	**ENABLED**, DISABLED	Removes 2-lane Rx logic when enabled.
MIPI_RX_2L_FIXED	ENABLED, **DISABLED**	Removes 4-lane Rx logic when enabled.
DS_2TO1_ALWAYS_ON	**ENABLED**, DISABLED	Removes non-downscale path when enabled.
DS_2TO1_ALWAYS_OFF	ENABLED, **DISABLED**	Removes 2:1 downscaler when enabled.
PLL_CONFIG	ENABLED, **DISABLED**	Implements APB interface when enabled.
MINI_ISP	**ENABLED**, DISABLED	Removes mini ISP module when disabled.
ST_OSD	ENABLED, **DISABLED**	Removes static OSD logic when disabled.
MAX_ML_IMG_WIDTH	Maximum: 480, **384**	Specifies maximum ML image width.
MAX_ML_IMG_SIZE	Maximum: 480 × 360, **384 × 288**	Specifies maximum ML image size.
MIPI_TX_4L_FIXED	**ENABLED**, DISABLED	Removes 1-lane and 2-lane logic when enabled.
MIPI_TX_2L_FIXED	ENABLED, **DISABLED**	Removes 1-lane and 4-lane logic when enabled.
MIPI_TX_1L_FIXED	ENABLED, **DISABLED**	Removes 2-lane and 4-lane logic when enabled.
MIPI_PASS_THRU	**ENABLED**, DISABLED	Removes pass-through logic when disabled.
MIPI_PASS_THRU_ONLY	ENABLED, **DISABLED**	Removes internal image pass when enabled.
DYN_OSD	ENABLED, **DISABLED**	Removes dynamic OSD logic will be removed when disabled.
MIPI_META	**ENABLED**, DISABLED	MIPI Meta Data logic when disabled.
P2B_READ_DELAY	4, **205**	Use 205 for 4-lane Rx/Tx with 720 Mbps input.
DEBUG_REGISTER	ENABLED, **DISABLED**	Adds debug registers to APB registers when enabled.
OSC_FOR_FPS_COUNT	ENABLED, **DISABLED**	Instantiates 32 kHz internal oscillator when this and DEBUG_REGISTER are both enabled.

When all Rx and Tx lanes are allowed and all OSD features are enabled, LUT usage is approximately 17k with 50 EBRs, supporting 8 bounding boxes and 16 text character overlays.

The compile time options are available in file:
mod_top_torna_revb_cpnx100_radiant/mod_top_torna_revb_cpnx100_radiant/remote_files/include/cpnx_video_path_directives.vh

Note: One bounding box overlay requires approximately 250 LUTs.

Examples:

- A fixed design with Rx 4-lane and Tx 4-lane can reduce LUT usage by approximately 2k and EBR usage by 2.
- A fixed design with Rx 2-lane and Tx 2-lane without the 2:1 downscaler can reduce LUT usage by approximately 3k and EBR usage by 12 compared to the full design.

4.3.3. Common APB Subordinate

The following table shows the APB register table shared by the three submodules.

Table 4.4. APB Register Map of Application Subsystem

Offset	Item	Bits	Default	Description
0x00	MIPI Output Mode	1:0	1	00: No MIPI output 01: Normal image out 10: Pass-through 11: ML image out (for debug)
	2:1 Downscaler Enable	2	0	Enables 2:1 downscaler (recommended when MIPI Rx is in 4-lane configuration).
	Number of MIPI Rx Data Lane	4	0	1: 4 lanes 0: 2 lanes
	Reserved	5	0	Reserved bits.
	Number of MIPI Tx Data Lane	7:6	1	3: 4 lanes

Offset	Item	Bits	Default	Description
				1: 2 lanes 0: 1 lane
	Input Bayer Format	9:8	0	0: RGGB 1: GRBG 2: GBRG 3: BGGR
	Output Bayer Format	11:10	0	0: RGGB 1: GRBG 2: GBRG 3: BGGR
	Keep MIPI Clock in HS During HB	12	0	1: Keep MIPI Tx Clock in HS mode during horizontal blanking. 0: Clock goes into LP mode.
	Read Delay from P2B FIFO	24:16	4	Set a value larger than 4 when byte clock bandwidth exceeds pixel clock bandwidth. Use THRESHOLD value shown in Byte-to-Pixel IP GUI.
0x04	ML Enable	0	1	1: ML enable 0: ML off
	ML RGB Enable	1	1	1: ML by RGB data 0: ML by grayscale data
	Static OSD Enable	4	1	Static OSD is active when enabled.
	Reserved	7:5	0	Reserved bits.
	Meta Data Data Type	13:8	0x30	Must be 0x12 or 0x30–0x37.
	Output Image WC	31:16	2,400 (decimal)	Horizontal pixel count × 1.25 for RAW10, maximum: 2,400 (decimal). Ignored in MIPI pass-through mode.
0x08	Input Horizontal Resolution / 2 – 1	10:0	959 (decimal)	(Input Horizontal Resolution / 2) – 1, maximum: 1,919.
	Input Vertical Resolution – 2	27:16	1,078 (decimal)	Input Vertical Resolution – 2, maximum: 2,878.
0x0C	Horizontal Trimming from Left	7:0	0 (decimal)	Maximum: 254 (only even numbers allowed).
	Vertical Trimming from Top	15:8	0 (decimal)	Maximum: 255 (even number recommended).
0x10	Horizontal Resolution After Cropping	11:0	1,920 (decimal)	Must not exceed *Input Horizontal Resolution – Horizontal Trimming from Left*.
	Vertical Resolution After Cropping	27:16	1,080 (decimal)	Must not exceed *Input Vertical Resolution – Vertical Trimming from Top*.
0x14	AXI Write Maximum Burst Length – 1	7:0	63 (decimal)	Maximum Burst Length – 1, for ML image write.
	AXI Read Maximum Burst Length – 1	15:0	63 (decimal)	Maximum Burst Length – 1, for static OSD pattern read.
0x18	ML Image Horizontal Resolution	8:0	480 (decimal)	Horizontal resolution in pixels, maximum: 480.
	ML Image Vertical Resolution	24:16	320 (decimal)	Vertical resolution in pixels, maximum: 320.
0x1C	Reserved	—	—	—
0x20	Address Offset of Static OSD	31:0	0x1000_0000	Static OSD address offset in HyperRAM.
0x24	Static OSD Size	17:0	15,360 (decimal)	Image Size (= horizontal size × vertical size), maximum: 131,072 (approximately = 1920 × 68.3).
0x28	Static OSD Left Position	10:0	0 (decimal)	Left edge position of OSD.
	Static OSD Top Position	26:16	1,072 (decimal)	Top edge position of OSD.
0x2C	Static OSD Right Position – 1	10:0	1,918 (decimal)	Right edge position of OSD – 1.
	Static OSD Bottom Position	26:16	1,079 (decimal)	Bottom edge position of OSD.
0x30	Static OSD Color	23:0	0xFF_FF_FF	bit[23:16]: B value bit[15:8]: G value bit[7:0]: R value
0x40	GPLL Lock	0	1	—

Offset	Item	Bits	Default	Description
	Rx FIFO Underflow	4	0	Cleared by writing 1.
	Rx FIFO Overflow	5	0	Cleared by writing 1.
	Rx FIFO Delay Error	6	0	Cleared by writing 1.
	ISP DS FIFO Underflow	8	0	Cleared by writing 1.
	ISP DS FIFO Overflow	9	0	Cleared by writing 1.
	P2B FIFO Underflow	10	0	Cleared by writing 1.
	P2B FIFO Overflow	11	0	Cleared by writing 1.
	Pass Through FIFO Underflow	12	0	Cleared by writing 1.
	Pass Through FIFO Overflow	13	0	Cleared by writing 1.
0x50	Static OSD Transfer Request	0	0	Cleared when transfer done signal is asserted.
	Dynamic OSD Transfer Request	1	0	Cleared when transfer done signal is asserted.
	Meta Data Transfer Request	2	0	Cleared when transfer done signal is asserted.
0x60	Static OSD Transfer Done	0	0	Cleared by writing 1.
	Dynamic OSD Transfer Done	1	0	Cleared by writing 1.
	Meta Data Transfer Done	2	0	Cleared by writing 1.
Debug Registers (Read Only – Implemented when DEBUG_REGISTER is Enabled)				
0x80	Rx Frame Rate	7:0	0	—
	Number of Long Packet per Frame	27:16	0	—
0x84	Number of Pixels per Line Go to ISP Sub	11:0	0	—
	Number of Lines per Frame Go to ISP Sub	27:16	0	—
0x88	Number of Pixels per Line in Output Sub	11:0	0	—
	Number of Lines per Frame in Output Sub	27:16	0	—
0x8C	Number of Tx Long Packets per Frame	11:0	0	—
	Cycle Length of tx_pkten	29:16	0	—

4.3.4. Input Submodule

The Input Submodule receives the video stream from the image sensor and performs RAW10-to-RGB conversion. This module also applies 2:1 downscaling when the image size is large (for example, a 4K image).

A PLL is instantiated in this module to generate various clocks for video path processing. Table 4.5 lists the various clock outputs, using 24 MHz as the reference clock.

Table 4.5. Input Submodule PLL

PLL Output	Description
clkop	Not in use.
clkos	Clock for Tx D-PHY transmitter.
clkos2	Clock for Tx D-PHY transmitter (90-degree shifted).
clkos3	Pixel clock.
clkos4	Rx byte clock.
clkos5	Pixel clock for downscaled image (same speed as clkos3 when 2:1 downscaled image is not applied).

Table 4.6 shows the APB register map for the PLL IP.

Table 4.6. APB Register Map of PLL IP

Offset	Item	Bits	Default	Description
0x00	Control Register	7:0	0x00	Write 0x01 after setting all other registers.
0x04–0xD4	Parameter Registers	7:0	0x03	Refer to Table 4.7 for more details.
			—	
			0x00	

Table 4.7. PLL Clock and APB Register Setting Examples

APB offset	PLL APB Register Values				
0x00	0x00	0x00	0x00	0x00	0x00
0x04	0x03	0x03	0x03	0x03	0x03
0x08	0x08	0x08	0x08	0x08	0x08
0x0C	0x00	0x00	0x00	0x00	0x00
0x10	0x00	0x00	0x00	0x00	0x00
0x14	0x80	0x80	0x80	0x80	0x80
0x18	0x00	0x00	0x00	0x00	0x00
0x1C	0x00	0x00	0x00	0x00	0x00
0x20	0x03	0x03	0x03	0x03	0x03
0x24	0x02	0x02	0x02	0x02	0x02
0x28	0x00	0x00	0x00	0x00	0x00
0x2C	0x0C	0x0C	0x0C	0x0C	0x0C
0x30	0x10	0x10	0x10	0x10	0x10
0x34	0x80	0x80	0x80	0x80	0x80
0x38	0x00	0x00	0x00	0x00	0x00
0x3C	0x0C	0x0C	0x0C	0x0C	0x0C
0x40	0x05	0x05	0x05	0x05	0x05
0x44	0x00	0x00	0x00	0x00	0x00
0x48	0x00	0x00	0x00	0x00	0x00
0x4C	0x00	0x00	0x00	0x00	0x00
0x50	0x00	0x00	0x00	0x00	0x00
0x54	0x00	0x00	0x00	0x00	0x00
0x58	0x00	0x00	0x00	0x00	0x00
0x5C	0x00	0x00	0x00	0x00	0x00
0x60	0x00	0x00	0x00	0x00	0x00
0x64	0x80	0x80	0x80	0x80	0x80
0x68	0x67	0x67	0x67	0x67	0x67
0x6C	0xEC	0xEC	0xEC	0xEC	0xEC
0x70	0xD3	0xD3	0xD3	0xD3	0xD3
0x74	0x7F	0x7F	0x7F	0x7F	0x7F
0x78	0x1E	0x1E	0x1E	0x1E	0x1E
0x7C	0x03	0x03	0x03	0x03	0x03
0x80	0x18	0x18	0x18	0x18	0x18
0x84	0x00	0x00	0x00	0x00	0x00
0x88	0x0C	0x00	0x40	0x00	0x00
0x8C	0x00	0x00	0x00	0x00	0x00
0x90	0xC0	0xD8	0xD8	0xD8	0xD8
0x94	0x09	0x0D	0x2D	0x1D	0x0D
0x98	0x04	0x08	0x1C	0x12	0x08

APB offset	PLL APB Register Values				
0x9C	0x8E	0x89	0x8B	0x89	0x89
0xA0	0x85	0xC7	0xC7	0xC7	0xC7
0xA4	0x03	0x69	0x69	0x69	0x69
0xA8	0x27	0x37	0xB7	0x77	0x37
0xAC	0x10	0x18	0x58	0x38	0x18
0xB0	0x38	0x24	0x24	0x24	0x24
0xB4	0x16	0x1E	0x1E	0x1E	0x1E
0xB8	0x0E	0x27	0x27	0x27	0x27
0xBC	0x00	0x00	0x00	0x00	0x00
0xC0	0x00	0x00	0x00	0x00	0x00
0xC4	0x00	0x00	0x00	0x00	0x00
0xC8	0x00	0x00	0x00	0x00	0x00
0xCC	0x88	0xB8	0xB8	0xB8	0xB8
0xD0	0x03	0x03	0x03	0x03	0x03
0xD4	0x00	0x00	0x00	0x00	0x00

Table 4.8 shows clock frequency examples under several configurations.

Table 4.8. PLL Clock Frequency

Number of Rx Lanes	2	4	4	4	4
Rx Bandwidth / Lane (Mbps)	912	720	720	720	720
Number of Tx Lanes	2	4	4	2	1
MIPI Pass-Through	DC	Yes	No	No	No
Tx Bandwidth / Lane (Mbps)	912	720	240	360	720
Rx Image Resolution	1,920 × 1,080	3,280 × 2,464	3,280 × 2,464	3,280 × 2,464	3,280 × 2,464
Tx Image Resolution	1,920 × 1,080	3,280 × 2,464	1,640 × 1,232	1,640 × 1,232	1,640 × 1,232
VCO	912	1,440	1,440	1,440	1,440
Clkop (MHz)	24	24	24	24	24
clkos (tx_pll_clk) (MHz)	456	360	120	180	360
clkos2 (tx_pll_clk_shift) (MHz)	456	360	120	180	360
clkos3 (clk_pixel; 2ppc) (MHz)	91.2	144	144	144	144
clkos4 (clk_byte_fr) (MHz)	114	90	90	90	90
clkos5 (clk_ds; 2ppc) (MHz)	91.2	36	36	36	36

The Rx D-PHY IP has its own APB register map, shown below in Table 4.9.

Table 4.9. APB Register Map of Rx D-PHY IP

Offset	Item	Bits	Default	Description
0x28	Number of MIPI Rx Data Lane – 1	2:1	3	4 lanes (2'b11) and 2 lanes (2'b01) are supported
0x9C	Data Type	5:0	0x2B	RAW10 is supported
0xD8	Data Settle Cycle	7:0	10	Depends on bandwidth. Refer to Table 4.10 for more details.

Table 4.10. APB Register Value for Settle Cycle of Rx D-PHY IP

APB Offset	Item	MIPI RX Bandwidth per lane (Mbps)								
		912	810	720	640	432	378	360	202.5	180
0xD8	Settle cycle based on lane speed	0x0A	0x09	0x08	0x07	0x03	0x02	0x02	0x00	0x00

4.4. ISP Submodule

The ISP Submodule contains the mini ISP and the static OSD controller.

4.4.1. Mini ISP

The high-level block diagram of the mini ISP is shown in Figure 4.5.

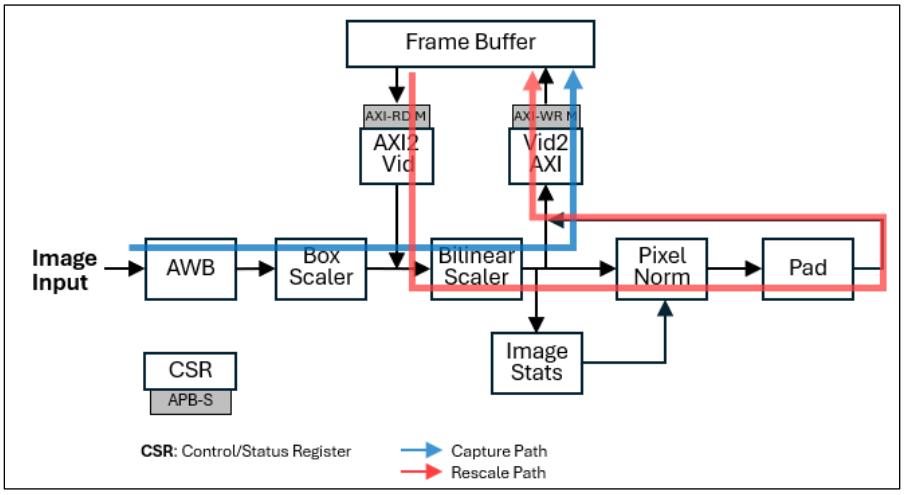

Figure 4.5. Mini ISP Capture and Rescale Flow

The functions of each block are described as follows:

- Automatic White Balance (AWB) performs automatic adjustment of color temperature to ensure colors, especially white, appear natural and accurate. This block is required only for RGB images and is not implemented for grayscale applications. AWB may be replaced with manual white balance depending on use cases.
- Box Scaler performs downscaling using an area/box algorithm, which applies simple averaging of an area into a single pixel. It can be configured for 1:2, 1:4, or 1:8 downscale factors.
- Bilinear Scaler supports cropping and downscaling of image size depending on ML use cases.
- Image Statistics calculates the minimum, maximum, and average pixel values of the captured image during the capture flow.
- Pixel Normalization performs global normalization on all pixels into the [0, 255] range and increases the brightness of dark areas in the image.
- Padding Block pads constant values on the left, right, top, or bottom of an image. This block is not implemented in Phase 1.
- Motion Detection Module can be enabled to detect motion for low-power requirements. It is disabled by default at compile time to save hardware resources if not needed.
- Control/Status Register contains the ISP register map for RISC-V usage.

4.4.2. Capture Flow

Image capture is performed by triggering the capture flow from RISC-V to directly scale the input image into the frame buffer. Figure 4.6 depicts an example register configuration for capturing a 640 × 360 grayscale image into the frame buffer from an input size of 1,920 × 1,080. This process involves setting the cropping and scaling configurations, frame buffer addresses, and other feature controls (for example, RUN_AS_GRAYSCALE) before triggering the capture start.

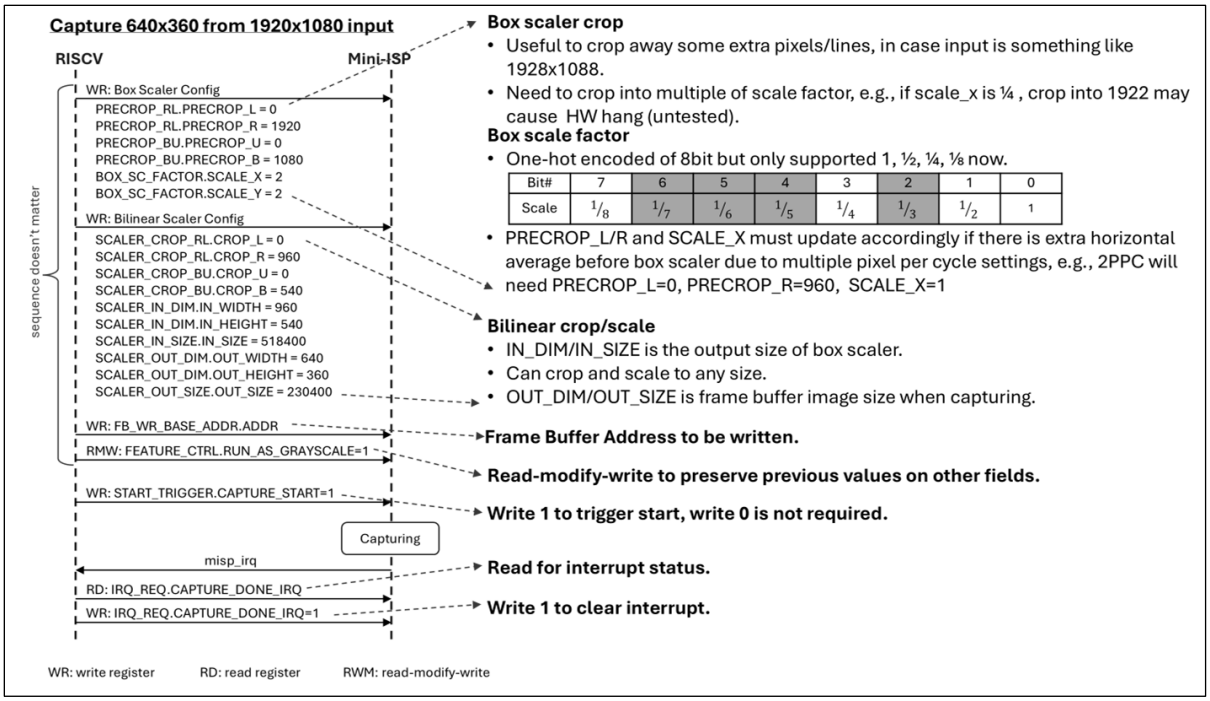

Figure 4.6. Grayscale Image Capture

4.4.3. Rescale Flow

In contrast, the rescale flow performs cropping and rescaling of an image from the frame buffer into an arbitrary size for ML processing. Figure 4.7 illustrates two examples of image rescaling to different output sizes.

The following example shows a RISC-V handshake to rescale a 640 × 360 grayscale image from the frame buffer into:

- 256 × 144 without cropping (left)
- 96 × 96 with cropping (right)

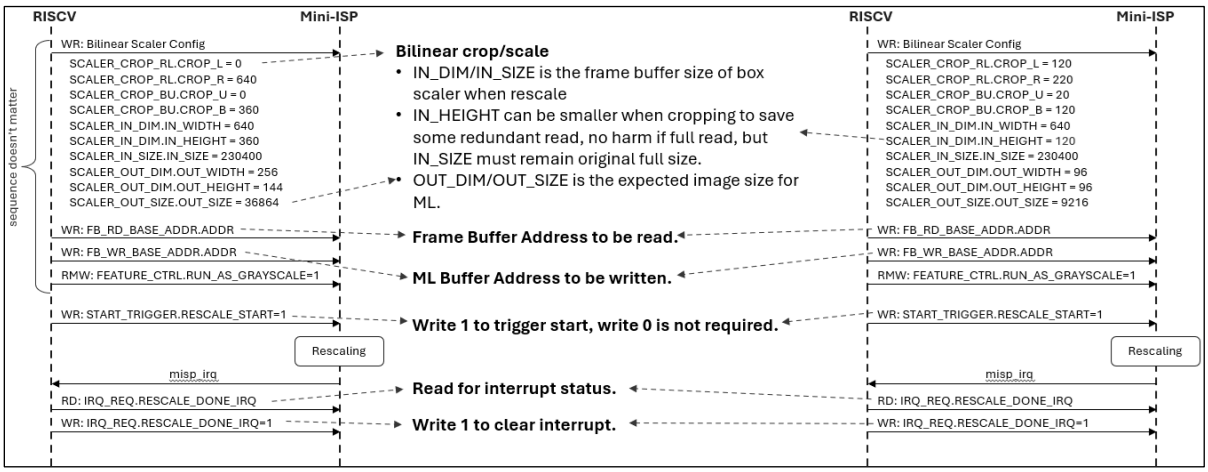

Figure 4.7. Rescale Flow Example

4.4.4. Static OSD

Note: The design is configurable to enable static OSD, dynamic OSD, and the metadata controller; however, the current design implementation does not use any of these features.

The static OSD feature is implemented through a compile-time option. The maximum size is 131,072 pixels as a single bitmap, which is approximately 1,920 × 68. Figure 4.8 shows a sample image with the maximum size of static OSD over a 1,080p image (gray box with stripes in the lower-right corner).

Figure 4.8. Static OSD Over 1,080p Image

Since it is static, the OSD block is expected to be enabled before video streaming and not dynamically enabled or disabled, even though the hardware can support enabling or disabling at frame boundaries.
The flow of the static OSD is as follows (steps 1 and 2 can be reversed):

1. Set the OSD information to HyperRAM.
2. Write the related information to the APB registers shown in Table 4.11.
3. Write 1 to the Static OSD Transfer Request bit in the APB register.
4. The FPGA reads OSD data from HyperRAM and stores the data in internal memory, then asserts st_osd_txfr_done_o as well as the OSD Transfer Done bit in the APB register.
5. Static OSD remains active as long as the enable bit (APB register bit[4] at offset 0x04) is set to 1.

4.4.5. Output Submodule

Output Submodule takes video data from ISP Submodule and applies dynamic OSD and meta-data insertion before outputting the video data on MIPI. The video output can be pass-through, internal video, or ML image data.

Tx D-PHY IP must be configured properly according to the given MIPI bandwidth. Refer to Table 4.12 for the configuration data.

Table 4.11. APB Register Map of Tx D-PHY IP

Offset	Item	Bits	Default	Description
0x28	Number of MIPI TX Data Lane – 1	2:1	3	4 lanes (2'b11), 2 lanes (2'b01), and 1 lane (2’b00) are supported.
0x7C	tLPx	7:0	—	The default value depends on the bandwidth. Refer to Table 4.12 for more details.
0x80	tCLK-PREP	7:0	—	
0x84	tCLK-HSZERO	7:0	—	
0x88	tCLKPRE	7:0	—	

Offset	Item	Bits	Default	Description
0x8C	tCLKPOST	7:0	—	
0x90	tCLKTRAIL	7:0	—	
0x94	tCLKEXIT	7:0	—	
0x98	tDAT-PREP	7:0	—	
0x9C	tDAT-HSZERO	7:0	—	
0xA0	tDATTRAIL	7:0	—	
0xA4	tDATEXIT	7:0	—	

Table 4.12. APB Register Values for Different Bandwidths of Tx D-PHY IP

APB Offset	Item	MIPI TX Bandwidth per lane									
		912 Mbps	810 Mbps	720 Mbps	640 Mbps	432 Mbps	378 Mbps	360 Mbps	240 Mbps	202.5 Mbps	180 Mbps
0x7C	tLPx	0x06	0x06	0x05	0x04	0x03	0x03	0x03	0x02	0x02	0x02
0x80	tCLK-PREP	0x05	0x04	0x04	0x04	0x03	0x02	0x02	0x02	0x01	0x01
0x84	tCLK-HSZERO	0x1E	0x1B	0x18	0x15	0x0F	0x0D	0x0C	0x08	0x07	0x06
0x88	tCLKPRE	0x02	0x02	0x02	0x02	0x02	0x02	0x02	0x02	0x02	0x02
0x8C	tCLK-POST	0x0D	0x0D	0x0C	0x0C	0x0A	0x0A	0x0A	0x09	0x09	0x08
0x90	tCLK-TRAIL	0x08	0x08	0x07	0x06	0x05	0x04	0x04	0x03	0x03	0x03
0x94	tCLK-EXIT	0x0C	0x0B	0x09	0x08	0x06	0x05	0x05	0x03	0x03	0x03
0x98	tDAT-PREP	0x06	0x05	0x05	0x04	0x03	0x03	0x03	0x02	0x02	0x02
0x9C	tDAT-HSZERO	0x0D	0x0C	0x0B	0x0A	0x07	0x06	0x06	0x04	0x04	0x04
0xA0	tDAT-TRAIL	0x0A[1] (0E)	0x09[1] (0D)	0x08[1] (0C)	0x07[1] (0B)	0x05[1] (09)	0x04[1] (08)	0x04[1] (08)	0x03[1] (07)	0x03[1] (07)	0x02[1] (06)
0xA4	tDAT-EXIT	0x0C	0x0B	0x09	0x08	0x06	0x05	0x05	0x03	0x03	0x03

Note:

1. This value is modified from the value provided by the Radiant IP user interface to avoid timing errors observed during simulation.

4.4.6. Dynamic OSD

Note: The design is configurable to enable static OSD, dynamic OSD, and the metadata controller; however, the current design implementation does not use any of these features.

Dynamic OSD provides two features: bounding-box overlay and text rendering. The maximum number of bounding boxes and text characters is defined by compile-time options for optimal logic generation.

Figure 4.9 shows a sample image for dynamic OSD. The line width of each bounding box is three pixels per line, and one character occupies 16 × 16 pixels.

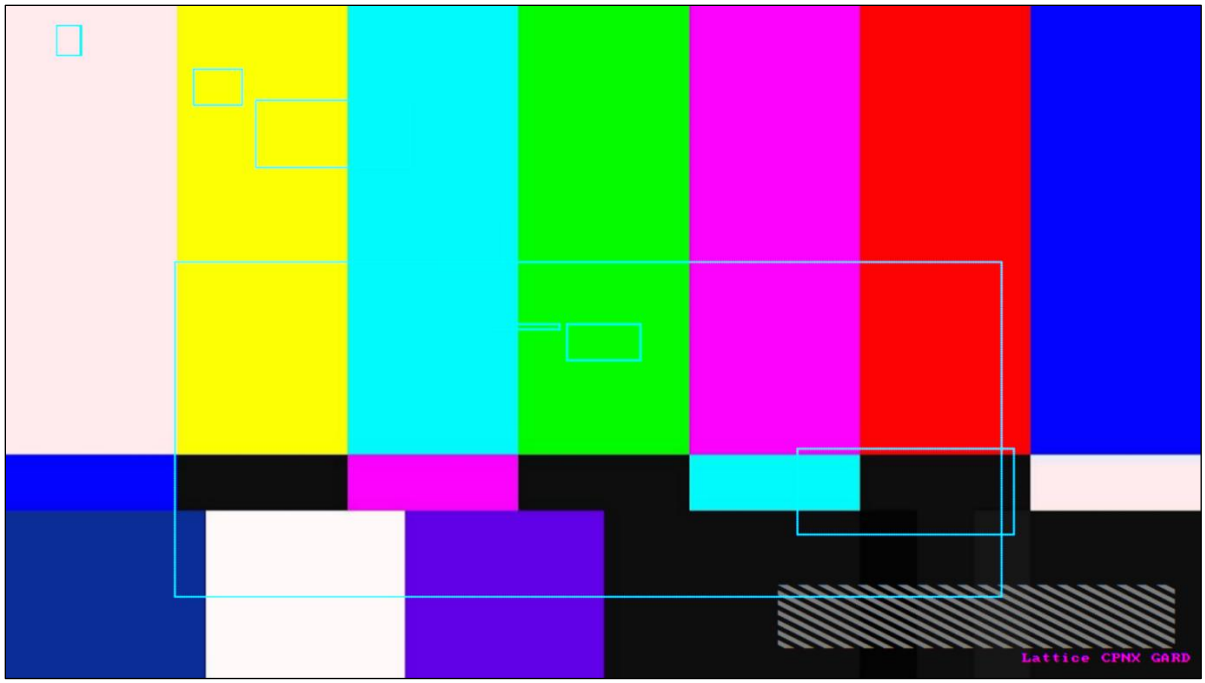

Figure 4.9. Sample Image for Dynamic OSD

Currently, up to 23 bounding boxes and 52 characters are supported. These limits can be extended by modifying the register map (one bounding box consumes approximately 250 LUTs, and text rendering supports only a single line).

The operative flow of the dynamic OSD is as follows:

1. Set the OSD information in the APB registers listed in Table 4.13. Set the number of B-Box and/or characters to 0 when no OSD is required.
2. Write 1 to the Dynamic OSD Transfer Request bit in the APB register.
3. The FPGA reads OSD data from the APB registers and stores the data in internal registers, then asserts dyn_osd_txfr_done_o and sets the Dynamic OSD Transfer Done bit in the APB register.
4. Dynamic OSD is applied in the next frame period.

Table 4.13. APB Register Map of Dynamic OSD

Offset	Item	Bits	Default	Description
0x00	Number of B-Box	4:0	0	Maximum: 23 (can be increased by expanding the register map, must not exceed the setting value in compile option)
0x04	Color of B-Box Line	23:0	0x00_FF_FF	RGB from LSByte.
0x08	Left Edge of B-Box #1	10:0	0	Minimum: 0 (must be smaller than Right Edge value – 6)
	Top Edge of B-Box #1	26:16	0	Minimum: 0 (must be smaller than Bottom Edge value – 6)
0x0C	Right Edge of B-Box #1	10:0	0	Maximum = Horizontal Resolution of Output Image – 1
	Bottom Edge of B-Box #1	26:16	0	Maximum = Vertical Resolution of Output Image – 1
0x10–0x14	Coordinates of B-Box #2	—	0	Same restrictions as B-Box #1.
0x18–0x1C	Coordinates of B-Box #3	—	0	Same restrictions as B-Box #1.
0x20–0x14	Coordinates of B-Box #4	—	0	Same restrictions as B-Box #1.
0x28–0x2C	Coordinates of B-Box #5	—	0	Same restrictions as B-Box #1.
0x30–0x34	Coordinates of B-Box #6	—	0	Same restrictions as B-Box #1.
0x38–0x3C	Coordinates of B-Box #7	—	0	Same restrictions as B-Box #1.
0x40–0x44	Coordinates of B-Box #8	—	0	Same restrictions as B-Box #1.
...				
0xB8–0xBC	Coordinates of B-Box #23	—	0	Same restrictions as B-Box #1.

Offset	Item	Bits	Default	Description
0xC0	Number of Characters	5:0	0	Maximum: 52 (can be increased by expanding the register map, must not exceed the setting value in compile option)
0xC4	Color of Text	23:0	0xFF_FF_FF	RGB from LSByte.
0xC8	X Position of the 1st Character – 1	10:0	0	LSB 4 bits ignored. Maximum = Horizontal Resolution of Output Image – (16 × Number of Characters) – 1
	Y Position of the 1st Character	26:16	0	LSB 4 bits ignored. Maximum = Vertical Resolution of Output Image – 16
0xCC	ASCII Codes of Character #1	6:0	0	7-bit ASCII codes
	ASCII Codes of Character #2	14:8	0	7-bit ASCII codes
	ASCII Codes of Character #3	22:16	0	7-bit ASCII codes
	ASCII Codes of Character #4	30:24	0	7-bit ASCII codes
0xD0	ASCII Codes of Characters #5–#8	—	0	7-bit ASCII codes × 4
0xD4	ASCII Codes of Characters #9–#12	—	0	7-bit ASCII codes × 4
0xD8	ASCII Codes of Characters #13–#16	—	0	7-bit ASCII codes × 4
0xDC	ASCII Codes of Characters #17–#20	—	0	7-bit ASCII codes × 4
…				
0xFC	ASCII Codes of Characters #49–#52	—	0	7-bit ASCII codes × 4

4.5. RISC-V Subsystem

The RISC-V is a 32-bit real-time processor core that acts as the central controller with full visibility of all system components, including peripherals, memory components, and sensors. It boots from TCM for deterministic, high-speed startup, ensuring low-latency access for critical routines and boot code.

4.5.1. Interconnect Hierarchy

- APB Interconnect:
  - Designed for low-speed peripherals such as UART, I2C, GPIO, and register bank.
  - Hosts multiple controller interfaces for configuring the video pipeline and ISP.
- South AXI Interconnect:
  - Connects the APB domain to the high-speed AXI domain, enabling access to memory.
  - Provides scalability for metadata handling and advanced video features.

4.5.2. Peripheral Subsystem

- UART – Used for debug and communication.
- I2C initiator – Configures the camera sensor (for example, camera setup and exposure settings).
- I2C target – Enables interaction with the Raspberry Pi host.
- GPIO – Provides general control signals for system management.
- Register bank – Stores configuration and status registers for video and ISP pipelines.

4.5.3. Video and ISP Configuration

- APB requestors configure video pipeline blocks and ISP pipeline for image processing tasks.
- Enables flexible control of image capture, enhancement, and streaming.

4.5.4. Memory Resources

- 128 kB TCM – Used for boot code and performance-critical routines.
- 16 kB AXI Metadata Buffer – Currently unused and reserved for purposes such as storing frame metadata and AI inference data.
- 8 MB HyperRAM – Used for run-time loading of the RISC-V RX CPU firmware.

4.5.5. Interrupt Mapping to RISC-V

Table 4.14. Interrupt Mapping to RISC-V

RISC-V Port name	Source (Peripheral IP)
IRQ_2	ML Done
IRQ_3	OSPI
IRQ_4	I2C_target
IRQ_5	UART
IRQ_6	I2C_initiator
IRQ_7	GPIO_0 (This IP drives MIPI Camera Enable)
IRQ_8	GPIO_1 (This IP drives Torna Board version control registers)
IRQ_9	GPIO_2 (This IP drives XO3D)
IRQ_10	GPIO_3 (This is connected to RPi)
IRQ_12	Scaler_IRQ (Capture_Done or Copy Done)

For more information about the platform level interrupt controller information, refer to the Platform Level Interrupt Controller section in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02298).

4.6. ML Subsystem

4.6.1. ML Core

The Lattice Advanced CNN Accelerator IP core is a computation engine that uses a deep neural network with fixed-point weights. It calculates neural network layers such as convolution, pooling, batch normalization, and fully connected layers by executing a sequence of firmware instructions with weight values. This firmware, called ML Firmware, is generated by the Lattice sensAI Neural Network Compiler.

For more information about the ML IP, refer to the Advanced CNN IP User Guide (FPGA-IPUG-02224).

The top-level external interfaces of this Advanced CNN IP include:

- Clock and reset
- Control and status
- Input interface (LMMI)
- Memory interface (AXI4)
- Result and debug-output interface

The following figure shows the block diagram of the Advanced CNN IP.

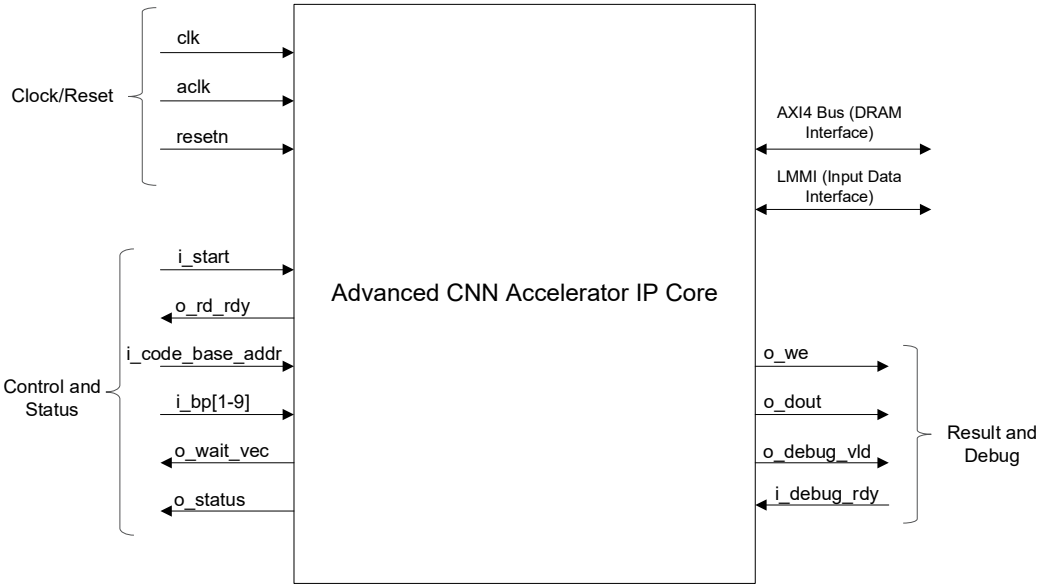

Figure 4.10. Advanced CNN Interfaces

The ML IP has the following three data transfer output interfaces:

- AXI interface
- LMMI interface
- Result interface

4.6.1.1. AXI Interface

This interface is used by the ML IP to access external DRAM or XIP (whichever is available) and execute command codes. During execution, the Advanced CNN Accelerator IP reads command codes over the AXI bus and performs calculations using internal execution engines. Intermediate data may be transferred to or from DRAM per command code. This 64-bit interface supports a configurable maximum burst length (32 or 256) for AXI4.

4.6.1.2. LMMI Interface

The input data interface is LMMI. Refer to the Lattice Memory Mapped Interface and Lattice Interrupt Interface (FPGA-UG-02039) for more details and timing diagrams.

If input data is small enough to fit in internal memory, writing to DRAM and reading back may waste cycles and energy. In such cases, external logic should write input data directly to the internal memory of the Advanced CNN Accelerator IP during idle state.

The read latency from request to data availability is up to four clock cycles. This interface supports configurable LMMI access modes: byte, half-word (16-bit), or word (32-bit).

4.6.1.3. Result Interface

The final blob data of the neural network and debug information are output to external logic through the result and debug interface.

When o_we is asserted, the o_dout signal contains valid final blob data. When o_debug_vld is asserted, the o_dout signal contains valid debug information. The IP continues transmitting available debug information while i_debug_rdy is asserted.

The IP halts until all debug information is transmitted.

4.6.2. ML Firmware Execution Flow

The ML event action workflow is shown in the following figure.

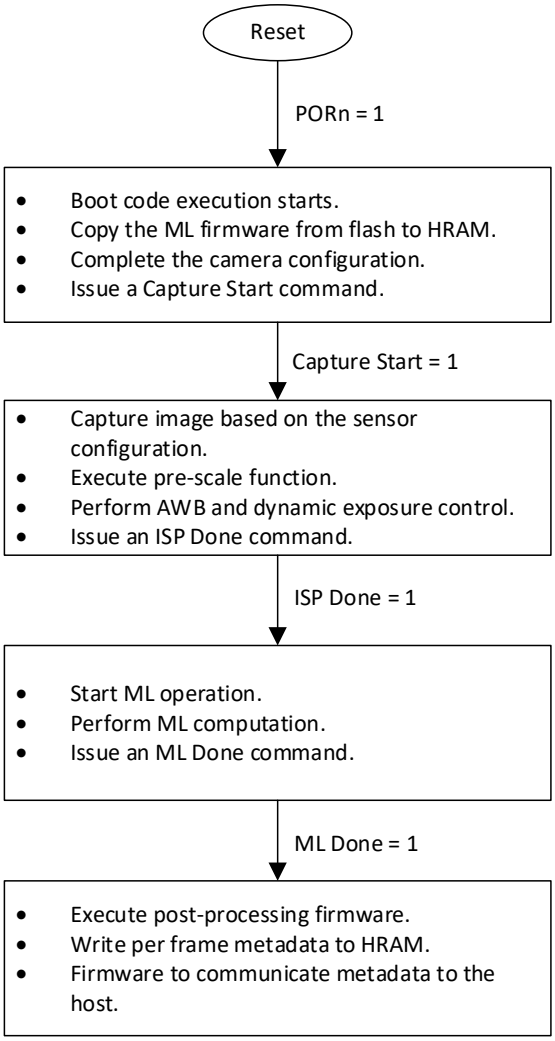

Figure 4.11. ML FW Execution Flow

4.6.3. External Memory Interface (HyperRAM)

The HyperRAM controller provides access to on-board HyperRAM devices through an AXI4 subordinate interface.

4.6.4. HyperRAM Devices

The SOM may have one or two 64 Mbit Infineon HyperRAM devices, providing a total of either 8 MB or 16 MB. The current architecture supports two HyperRAM devices, each with 8 MB capacity.

4.6.5. HyperRAM Memory Controller

The HyperRAM controller supports both single and dual memory device configurations.

In either single-chip or dual-chip mode, the HyperRAM controller supports a 64-bit wide AXI data path, ensuring compatibility with the Advanced CNN's 64-bit AXI interface without requiring data width conversion. In single-chip mode, the controller optionally supports 32-bit or 64-bit AXI data width. For best performance, 64-bit mode is recommended regardless of the number of memory ICs.

4.7. System Address Map and Views

4.7.1. View from RISC-V

Table 4.15. Address Map View from RISC-V

Target	Start Address	End Address	Size
TCM	0x0000_0000	0x0001_FFFF	128 kB
Flash Memory[1]	0x0800_0000	0x0BFF_FFFF	64 MB
Metadata buffer	0x0C00_0000	0x0C00_3FFF	16 kB
HyperRAM2	0x0C80_0000	0x0CFF_FFFF	8 MB
GPIO0	0x4000_0000	0x4000_0FFF	4 kB
I2C Target	0x4000_1000	0x4000_1FFF	4 kB
UART	0x4000_2000	0x4000_2FFF	4 kB
GARD Register Bank	0x4000_3000	0x4000_3FFF	4 kB
I2C Controller	0x4000_4000	0x4000_4FFF	4 kB
Octal SPI Controller	0x4000_6000	0x4000_6FFF	4 kB
GPIO1	0x4000_7000	0x4000_7FFF	4 kB
GPIO2	0x4000_8000	0x4000_8FFF	4 kB
GPIO3	0x4000_9000	0x4000_9FFF	4 kB
Video Pipeline Common APB	0x4000_A000	0x4000_AFFF	4 kB
Video Pipeline GPLL APB	0x4000_B000	0x4000_BFFF	4 kB
Video Pipeline Rx D-PHY APB	0x4000_C000	0x4000_CFFF	4 kB
Video Pipeline Tx D-PHY APB	0x4000_D000	0x4000_DFFF	4 kB
Video Pipeline mini-ISP APB	0x4001_0000	0x4001_FFFF	64 kB
HyperRAM1	0x8000_0000	0x807F_FFFF	8 MB
CLINT	0xF200_0000	0xF20F_FFFF	1 MB
PLIC	0xFC00_0000	0xF23F_FFFF	4 MB

Note:

1. XIP is enabled

4.7.2. View from ML IP

Table 4.16. Address Map View from ML IP

Target	Start Address	End Address	Size
HyperRAM1	0x8000_0000	0x807F_FFFF	8 MB

4.7.3. View from Mini ISP

Table 4.17. Address Map View from Mini ISP

Target	Start Address	End Address	Size
HyperRAM1	0x8000_0000	0x807F_FFFF	8 MB

4.8. GARD Register Bank

The following registers are available in the GARD regbank.

Table 4.18. GARD regbank

Name	Offset Address	Access	Field	Bit Index	Reset Value
Platform Registers					
Reset controller	0x0000	RW	perip_resetn	[1:1]	0x1
			ml_resetn	[0:0]	0x1
Scaler Registers[1]					
Scaler crop right	0x0200	RW	sc_crop_rl_crop_r	[15:0]	0x0000
Scaler crop left	0x0204	RW	sc_crop_rl_crop_l	[15:0]	0x0000
Scaler crop bottom	0x0208	RW	sc_crop_bu_crop_b	[15:0]	0x0000
Scaler crop up	0x020C	RW	sc_crop_bu_crop_u	[15:0]	0x0000
Scaler input dim width	0x0210	RW	sc_in_dim_width	[15:0]	0x0000
Scaler input dim height	0x0214	RW	sc_in_dim_height	[15:0]	0x0000
Scaler write base address	0x0218	RW	sc_wr_base_addr	[31:0]	0x0000_0000
Scaler read base address	0x021C	RW	sc_rd_base_addr	[31:0]	0x0000_0000
Scaler output size	0x0220	RW	sc_out_size	[31:0]	0x0000_0000
Scaler output dim width	0x0224	RW	sc_out_dim_width	[15:0]	0x0000
Scaler output dim height	0x0228	RW	sc_out_dim_height	[15:0]	0x0000
Scaler trigger rescale start	0x022C	RW	sc_trigger_rescale_start	[0:0]	0x0
Scaler trigger capture start	0x0230	RW	sc_trigger_capture_start	[0:0]	0x0
Scaler rescale interrupt clear	0x0234	RW	sc_rescale_irq_clr	[0:0]	0x0
Scaler capture interrupt clear	0x0238	RW	sc_capture_irq_clr	[0:0]	0x0
Scaler trigger ml read start	0x023C	RW	sc_trigger_ml_read_start	[0:0]	0x0
Scaler rescale done	0x0240	RO	sc_rescale_done	[0:0]	0x0
Scaler ml read done	0x0244	RO	sc_ml_read_done	[0:0]	0x0
Metadata Registers[2]					
Meta data 1	0x0400	RO	meta_data_1	[31:0]	0x0000_0000
Meta data 2	0x0404	RO	meta_data_2	[31:0]	0x0000_0000
Meta data 3	0x0408	RO	meta_data_3	[31:0]	0x0000_0000
Meta data 4	0x040C	RO	meta_data_4	[31:0]	0x0000_0000
Meta data 5	0x0410	RO	meta_data_5	[31:0]	0x0000_0000
Meta data 6	0x0414	RO	meta_data_6	[31:0]	0x0000_0000
Meta data 7	0x0418	RO	meta_data_7	[31:0]	0x0000_0000
Meta data 8	0x041C	RO	meta_data_8	[31:0]	0x0000_0000
ML Control and Statistics[3]					
ML stat cycles	0x0800	RO	ml_cycles	[31:0]	0x0000_0000
ML stat commands	0x0804	RO	ml_commands	[31:0]	0x0000_0000
ML stat dma cycles	0x0808	RO	ml_dma_cycles	[31:0]	0x0000_0000
ML stat dma commands	0x080C	RO	ml_dma_commands	[31:0]	0x0000_0000
ML stat loss time	0x0810	RO	ml_loss_time	[31:0]	0x0000_0000
ML stat eu cycles	0x0814	RO	ml_eu_cycles	[31:0]	0x0000_0000
ML stat fc cycles	0x0818	RO	ml_fc_cycles	[31:0]	0x0000_0000
ML stat ldma cycles	0x081C	RO	ml_ldma_cycles	[31:0]	0x0000_0000
ML stat ve cycles	0x0820	RO	ml_ve_cycles	[31:0]	0x0000_0000
ML stat scale cycles	0x0824	RO	ml_scale_cycles	[31:0]	0x0000_0000
ML stat wait cycles	0x0828	RO	ml_wait_cycles	[31:0]	0x0000_0000
ML stat wait dma d2e	0x082C	RO	ml_wait_dma_d2e	[31:0]	0x0000_0000

Name	Offset Address	Access	Field	Bit Index	Reset Value
ML stat wait dma e2d	0x0830	RO	ml_wait_dma_e2d	[31:0]	0x0000_0000
ML stat wait ldma s2l	0x0834	RO	ml_wait_ldma_s2l	[31:0]	0x0000_0000
ML stat wait ldma l2s	0x0838	RO	ml_wait_ldma_l2s	[31:0]	0x0000_0000
ML stat wait ml rd	0x083C	RO	ml_wait_ml_rd	[31:0]	0x0000_0000
ML stat wait ml wr	0x0840	RO	ml_wait_ml_wr	[31:0]	0x0000_0000
ML stat wait conv spd	0x0844	RO	ml_wait_conv_spd	[31:0]	0x0000_0000
ML stat wait ve spd	0x0848	RO	ml_wait_ve_spd	[31:0]	0x0000_0000
ML stat gpo	0x084C	RO	ml_gpo	[31:0]	0x0000_0000
ML stat status	0x0850	RO	ml_status	[10:0]	0x000
ML code base addr	0x0854	RW	ml_code_base_addr	[31:0]	0x8000_0000
ML bp1	0x0858	RW	ml_bp1	[31:0]	0x8000_0000
ML bp2	0x085C	RW	ml_bp2	[31:0]	0x8000_0000
ML bp3	0x0860	RW	ml_bp3	[31:0]	0x8000_0000
ML bp4	0x0864	RW	ml_bp4	[31:0]	0x8000_0000
ML bp5	0x0868	RW	ml_bp5	[31:0]	0x8000_0000
ML bp6	0x086C	RW	ml_bp6	[31:0]	0x8000_0000
ML bp7	0x0870	RW	ml_bp7	[31:0]	0x8000_0000
ML bp8	0x0874	RW	ml_bp8	[31:0]	0x8000_0000
ML bp9	0x0878	RW	ml_bp9	[31:0]	0x8000_0000
ML start	0x087C	RW	ml_start	[0:0]	0x0
ML rdy	0x0880	RO	ml_rdy	[0:0]	0x0
ML rdy interrupt clear	0x0884	RW	ml_rdy_irq_clr	[0:0]	0x0

Notes:

1. These registers are valid only for the HMI application.
2. These registers are not used.
3. Statistics values are available only if you instantiate the *ml_stat* block in the application subsystem.

4.9. GPIO Mapping

The following table shows the mapping of various memory-mapped GPIOs to the LFCPNX-100 pins.

Table 4.19. GPIO Mapping

GPIO	Pin	Source/Target
GPIO_0	0	Not connected on carried board
GPIO_0	1	Not connected on carried board
GPIO_0	2	GPIO pin of J12 MIPI connector
GPIO_0	3	GPIO pin of J12 MIPI connector
GPIO_0	5	J4 Header Pin 2
GPIO_0	6	J4 Header Pin 4
GPIO_0	7	J4 Header Pin 6
GPIO_0	8	J4 Header Pin 8
GPIO_1	[2:0]	Board version control
GPIO_2	0	MachXO3D pmu_wakeupn
GPIO_2	[3:0]	MachXO3D GPIOs
GPIO_3	0	NC
GPIO_3	1	NC
GPIO_3	2	GND
GPIO_3	3	TP44

GPIO	Pin	Source/Target
GPIO_3	4	TP42
GPIO_3	5	NC
GPIO_3	6	TP43
GPIO_3	7	NC
GPIO_3	8	RPI_CPNX_SCLK
GPIO_3	9	RPI_CPNX_CSN
GPIO_3	10	RPI_CPNX_MOSI
GPIO_3	11	RPI_CPNX_MISO
GPIO_3	14	FTDI_PWREN#
GPIO_3	15	FTDI_SUSPEND#
GPIO_3	16	RPI_PCIE_PWR_EN
GPIO_3	17	RPI_PCIe_nRST
GPIO_3	18	RPI_PCIe_CLK_nREQ
GPIO_3	19	RPI_PCIE_nWAKE
GPIO_3	20	RPI_CPNX_GPIO4
GPIO_3	21	RPI_CPNX_GPIO1
GPIO_3	22	RPI_CPNX_GPIO2
GPIO_3	23	RPI_CPNX_GPIO3
GPIO_3	24	RPI_CPNX_GPIO5

4.10. Pin Locking

The top level pins of the LFCPNX-100 FPGA used on the SOM Torna Rev B board are shown in following table.

Table 4.20. Pinouts

Pin	Bank	VCCIO (V)	Description	Function
M6	4	1.2	CPNX_CAM0_MIPI_CKN	MIPI camera input.
M7	4	1.2	CPNX_CAM0_MIPI_CKP	
K10	4	1.2	CPNX_CAM0_MIPI_D0N	
K11	4	1.2	CPNX_CAM0_MIPI_D0P	
K7	4	1.2	CPNX_CAM0_MIPI_D1N	
K8	4	1.2	CPNX_CAM0_MIPI_D1P	
L7	4	1.2	CPNX_CAM0_MIPI_D2N	—
L8	4	1.2	CPNX_CAM0_MIPI_D2P	—
L10	4	1.2	CPNX_CAM0_MIPI_D3N	—
M10	4	1.2	CPNX_CAM0_MIPI_D3P	—
L9	4	1.2	CPNX_CAM1_MIPI_CKN	MIPI output to the Raspberry Pi.
K9	4	1.2	CPNX_CAM1_MIPI_CKP	
P10	4	1.2	CPNX_CAM1_MIPI_D0N	
P9	4	1.2	CPNX_CAM1_MIPI_D0P	
R10	4	1.2	CPNX_CAM1_MIPI_D1N	
T10	4	1.2	CPNX_CAM1_MIPI_D1P	
T8	4	1.2	CPNX_CAM1_MIPI_D2N	—
R8	4	1.2	CPNX_CAM1_MIPI_D2P	—
N6	4	1.2	CPNX_CAM1_MIPI_D3N	—
N7	4	1.2	CPNX_CAM1_MIPI_D3P	—
H12	7	1.8	MIPI_CLK_EN	—
E1	1	3.3	TCK	RISC-V JTAG debugger and FPGA configuration.

Pin	Bank	VCCIO (V)	Description	Function
F5	1	3.3	TDI	
F3	1	3.3	TDO	
G7	1	3.3	TMS	
K2	2	3.3	clk_24m_i	24 MHz clock input from the MachXO3D device.
G14	7	1.8	cpnx_wrio[1]	Test points and NCs (connected to GPIO3).
G15	7	1.8	cpnx_wrio[2]	
E15	7	1.8	cpnx_wrio[3]	
E14	7	1.8	cpnx_wrio[4]	
H11	7	1.8	cpnx_wrio[5]	
H10	7	1.8	cpnx_wrio[6]	
F11	7	1.8	cpnx_wrio[7]	
F12	7	1.8	cpnx_wrio[8]	
G6	1	3.3	cpnx_wrio_3v3_l[1]	Pins connecting to the Raspberry Pi (connected to GPIO3).
G5	1	3.3	cpnx_wrio_3v3_l[2]	
G3	1	3.3	cpnx_wrio_3v3_l[3]	
G2	1	3.3	cpnx_wrio_3v3_l[4]	
H2	1	3.3	cpnx_wrio_3v3_m[10]	
G1	1	3.3	cpnx_wrio_3v3_m[11]	
H1	1	3.3	cpnx_wrio_3v3_m[12]	
F6	1	3.3	cpnx_wrio_3v3_m[13]	
E3	1	3.3	cpnx_wrio_3v3_m[14]	
D1	0	3.3	cpnx_wrio_3v3_m[15]	
C2	0	3.3	cpnx_wrio_3v3_m[16]	
C1	0	3.3	cpnx_wrio_3v3_m[17]	
H5	1	3.3	cpnx_wrio_3v3_m[7]	
H4	1	3.3	cpnx_wrio_3v3_m[8]	
H3	1	3.3	cpnx_wrio_3v3_m[9]	
F1	1	3.3	cpnx_xo3d_pmu_wakeupn	MachXO3D device (connected to GPIO2).
K16	5	1.8	hr1_ck_o	HyperRAM chip 1.
K15	5	1.8	hr1_ckn_o	
R16	5	1.8	hr1_csb_o	
N16	5	1.8	hr1_dq_io[0]	
M16	5	1.8	hr1_dq_io[1]	
P15	5	1.8	hr1_dq_io[2]	
N15	5	1.8	hr1_dq_io[3]	
P16	5	1.8	hr1_dq_io[4]	
M15	5	1.8	hr1_dq_io[5]	
L15	5	1.8	hr1_dq_io[6]	
L16	5	1.8	hr1_dq_io[7]	
P14	5	1.8	hr1_rstb_o	
K14	5	1.8	hr1_rwds_io	
N3	3	1.8	hr0_ck_o	HyperRAM chip 2.
M3	3	1.8	hr0_ckn_o	
T5	3	1.8	hr0_csb_o	
M4	3	1.8	hr0_dq_io[0]	
N5	3	1.8	hr0_dq_io[1]	
L5	3	1.8	hr0_dq_io[2]	
M5	3	1.8	hr0_dq_io[3]	

Pin	Bank	VCCIO (V)	Description	Function
N4	3	1.8	hr0_dq_io[4]	
L6	3	1.8	hr0_dq_io[5]	
T6	3	1.8	hr0_dq_io[6]	
R6	3	1.8	hr0_dq_io[7]	
K5	3	1.8	hr0_rstb_o	
R5	3	1.8	hr0_rwds_io	
J11	6	3.3	i2c_m_scl_io	I2C controller for camera configuration.
J10	6	3.3	i2c_m_sda_io	
F2	1	3.3	i2c_s_scl_io	I2C target for communication with the Raspberry Pi.
E2	1	3.3	i2c_s_sda_io	
J9	6	3.3	rpi_cam1_gpio_0	MIPI camera enable (connected to GPIO0).
J8	6	3.3	rpi_cam1_gpio_1	
J16	6	3.3	rpi_cam_gpio_0	
J15	6	3.3	rpi_cam_gpio_1	
J7	2	3.3	rstn_in	System reset from the MachXO3D device.
E5	0	3.3	spi_clk_o	Quad SPI.
D4	0	3.3	spi_css_n_o	
C3	0	3.3	spi_d2	
D2	0	3.3	spi_d3	
C4	0	3.3	spi_miso	
D3	0	3.3	spi_mosi	
D16	7	1.8	torna_ver_i[0]	Board version control (connected to GPIO1).
E12	7	1.8	torna_ver_i[1]	
E13	7	1.8	torna_ver_i[2]	
F9	1	3.3	uart_rxd_i	UART link with the Raspberry Pi or FTDI.
E6	1	3.3	uart_txd_o	
J6	2	3.3	xo3d_cpnx_gpio2	GPIOs from the MachXO3D device (connected to GPIO2).
J2	2	3.3	xo3d_cpnx_gpio4	
K1	2	3.3	xo3d_cpnx_gpio5	
J1	2	3.3	xo3d_cpnx_gpio6	

4.11. Resource Utilization

The LFCPNX-100 utilization for GARD with the application subsystem is provided in the following table. The resource utilization is based on the system configuration described below:

- LFCPNX-100 GARD:
  - ML IP: 4 LRAMs, 2 VE SPDs, VE ALU disabled
  - RISC-V RX CPU IP core in balanced configuration
  - OSPI controller for 4 lines with XIP enabled
  - 16 kB metadata buffer
  - 128 kB TCM using LRAMs
  - I2C and UART interfaces for host communication
  - I2C initiator for camera configuration
  - Two HyperRAM memory controllers
  - GPIOs and register bank
- Application Subsystem:
  - 4 MIPI lanes for input and output with resolution 3,280 × 2,464 at 30 FPS
  - 2:1 downscaler in input subsystem

- MIPI passthrough
- Bilinear and 2D scalers, AWB, and pixel normalization enabled in mini ISP

Table 4.21. FPGA Resource Utilization

Resources	Used	Available	Percentage
Registers	43,893	80,349	55 %
LUT4s	55,202	79,872	69 %
PIOs	142	159	89 %
Block RAMs	154	208	74 %
Large RAMs	6	7	86 %
Multipliers	103	156	66 %

4.12. Creating FPGA Bitstream File

4.12.1. MachXO3D

This section describes the procedure for generating the MachXO3D bitstream using the Lattice Diamond FPGA design software:

1. Launch the Lattice Diamond software tool.
2. Click **File** > **Open Project**. From the project database, open the Lattice Diamond project file (*.ldf*).
3. Click the **Process** tab at the bottom left of the pane and verify that the checkbox next to **JEDEC File** is checked, as shown in the figure below.

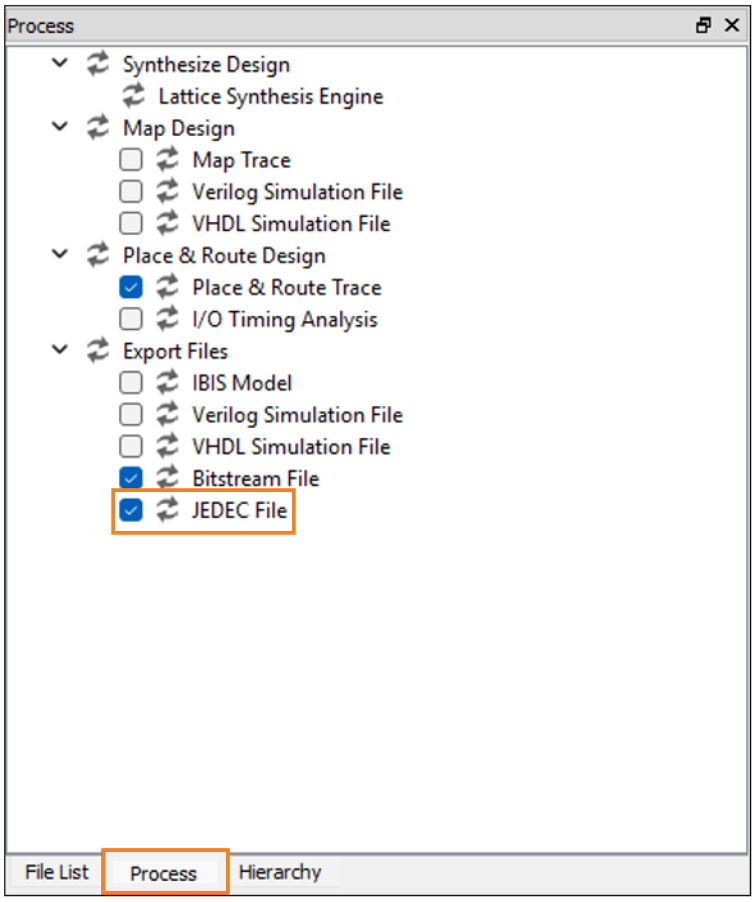

Figure 4.12. JEDEC File Generation

4. Click **Process** > **Run** or double-click **JEDEC File** and wait for the process to complete. A green checkmark over **JEDEC File** indicates completion, as shown in the figure below.

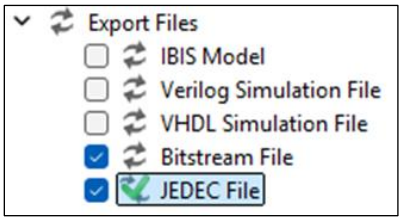

Figure 4.13. JEDEC File Creation Complete

4.12.2. CertusPro-NX (LFCPNX-100)

This section provides the procedure for creating the FPGA bitstream file using the Lattice Radiant software:

1. Launch the Lattice Radiant software tool.
2. Click **File** > **Open Project**.
3. From the project database, open the Lattice Radiant project file (*.rdf*) from the *mod_top_torna_revb_cpnx100_radiant/mod_top_torna_revb_cpnx100.rdf* folder.
4. Click **Export Files** to generate the bit file. View the log message in the output window for the generated bitstream. Find the generated .bit file in the *mod_top_torna_revb_cpnx100_radiant/impl_1* folder.

Figure 4.14. Bitstream Generation

5. At the end of bit file generation process, the TCL console displays the completion message, as shown in the figure below.

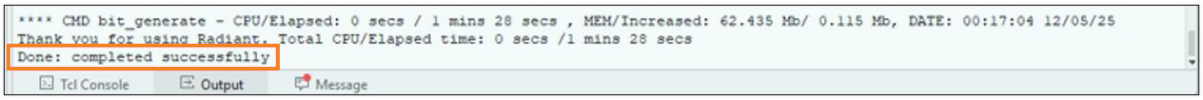

Figure 4.15. Generated Bitstream Log

4.13. Video Capture Flow for AI

For MOD and DD applications, the captured input image from the camera is stored in external memory (HyperRAM). The ML engine reads this captured image from external memory and writes the output back to external memory.

The design supports a two-step scaler flow, where the capture operation is performed first, followed by a rescale operation on the image stored in external memory. The GARD assumes the IMX219 camera with four MIPI lanes enabled. The input resolution of the camera is 3,280 × 2,464 at 30 FPS, and the same resolution is driven to the MIPI output toward the Raspberry Pi host.

The video capture flow is divided into the following three steps:

1. Capture the image.
2. Run the rescale flow.
3. Start the ML engine.

4.13.1.1. Capture Image

You need to configure the video pipeline registers. The details of these registers are available in the Application Subsystem section. The following GDB configuration example configures the video pipeline to passthrough a 3,280 × 2,464 resolution image at 4-lane MIPI input to MIPI output:

```
set {int}0x4000A000 = 0x03D6
set {int}0x4000A008 = 0x099E0667
set {int}0x4000A00C = 0x0
set {int}0x4000A010 = 0x09A00CD0
```

At this stage, it is assumed that GARD has started the camera. Next, configure the mini ISP to capture the image in external memory at address 0x80000000. The image stored in memory is 384 × 288 resolution:

```
# clear rescale/capture done
set {int}0x40010008 = 0x03

# config
set {int}0x40010010 = 0x80

# precrop_X, precrop_Y
set {int}0x40012100 = 0x03340000
set {int}0x40012104 = 0x04D00000

# box_scaler_factors
set {int}0x40012108 = 0x00000208

# in_dim, in size
set {int}0x40012118 = 0x019A0134
set {int}0x4001211C = 0x0001ED48

# crop_RL,crop_BU
set {int}0x40012110 = 0x019A0000
set {int}0x40012114 = 0x01340000

# out_dim, out_size
set {int}0x40012120 = 0x01800120
set {int}0x40012124 = 0x0001B000

# Burst length
set {int}0x40012130 = 0x00001F1F

# Write address
set {int}0x40012134 = 0x80000000

# Capture trigger
set {int}0x40010004 = 0x1
```

After the above capture configuration, the *capture_done* interrupt must be asserted.

4.13.1.2. Run Rescale Flow

Next, run the rescale flow on mini ISP to perform pixel normalization on the captured image. Optionally, you can apply further scaling in this step. The following example shows the GDB configuration sequence for reading a 384 × 288 input from HyperRAM at address 0x80000000 and writing a 384 × 288 image after pixel normalization into memory at address 0x80600000:

```
# clear rescale/capture done
set {int}0x40010008 = 0x03
```

```
# config
set {int}0x40010010 = 0x80

# in_dim, in size
set {int}0x40012118 = 0x01800120
set {int}0x4001211C = 0x0001B000

# crop_RL,crop_BU
set {int}0x40012110 = 0x01800000
set {int}0x40012114 = 0x01200000

# out_dim, out_size
set {int}0x40012120 = 0x01800120
set {int}0x40012124 = 0x0001B000

# Burst length
set {int}0x40012130 = 0x00001F1F

# Write address
set {int}0x40012134 = 0x80600000

# Read address
set {int}0x40012138 = 0x80000000

# Rescale trigger
set {int}0x40010004 = 0x2
```

4.13.1.3. Start ML Engine

After the rescaled image is available in external memory for the ML engine:

1. Define *ml_code_base_addr* to the address of external memory.
2. Issue *ml_start*.
3. The ML engine runs and, upon completion, asserts an ML interrupt signal. This interrupt can be cleared by writing to the *ml_rdy_irq_clr* register.

4.14. IP Configurations

Some parts of the GARD RTL design are created using the Lattice Propel Builder software, while other components are custom, such as hand-instantiated and integrated modules. The Lattice Propel Builder design provides easy modifications and scalability options, while the custom design is targeted for area-efficient applications.

To modify custom RTL, you must edit the corresponding header files or update the parameters directly in the RTL files.

4.14.1. IP Versions

The following table lists all IPs used in the GARD RTL along with their versions.

Table 4.22. IP Versions

Soft IP	Version
RISC-V RX CPU	2.6.0
Octal SPI Controller[1]	1.3.1
General Purpose I/O	1.8.0
UART	1.4.0
AXI4 Interconnect	2.2.0
AXI4 Mux	0.0.5
APB Interconnect	1.3.0

Soft IP	Version
AXI-to-APB Bridge	1.4.0
I2C Target	2.4.0
I2C Controller	2.3.0
AXI4 Sync	1.0.0
APB Sync	1.1.0
Advanced CNN	3.0.0
D-PHY Rx	2.0.0 (customized)
D-PHY Tx	2.3.0 (customized)
Byte-to-Pixel	Custom design
Pixel-to-Byte	1.9.0 (customized)
Down Scaler	Custom design
Tightly Coupled Memory (TCM)	1.5.2
HyperRAM Controller	0.2.5
OSC	1.4.0
PLL	1.9.1

Note:

1. This release uses the OSPI Controller internal version 1.3.1, which is packaged with the solution and available as a custom IPK. For any changes to the OSPI Controller IP configuration, use IP version 1.4.0, which is available on the IP server released with the Propel Builder tool version 2025.2.

Each component in the block diagram inside the LFCPNX-100 GARD box is instantiated using the IPs in the Propel Builder software.

The IP features and parameters are described in the following sections. The MOD and DD design package contains some custom-designed IPs that are not available on IP servers.

4.14.2. RISC-V RX CPU

The RISC-V RX CPU IP has AXI-based instruction and data ports. The instruction ports connect to memory that contains the bootloader software or the FreeRTOS application software for CPU execution. The data port connects to memory and peripherals for control.

For more information about the IP, refer to the RISC-V RX CPU IP User Guide (FPGA-IPUG-02298).

The following figures show the parameters selected and configured during IP instantiation in the design.

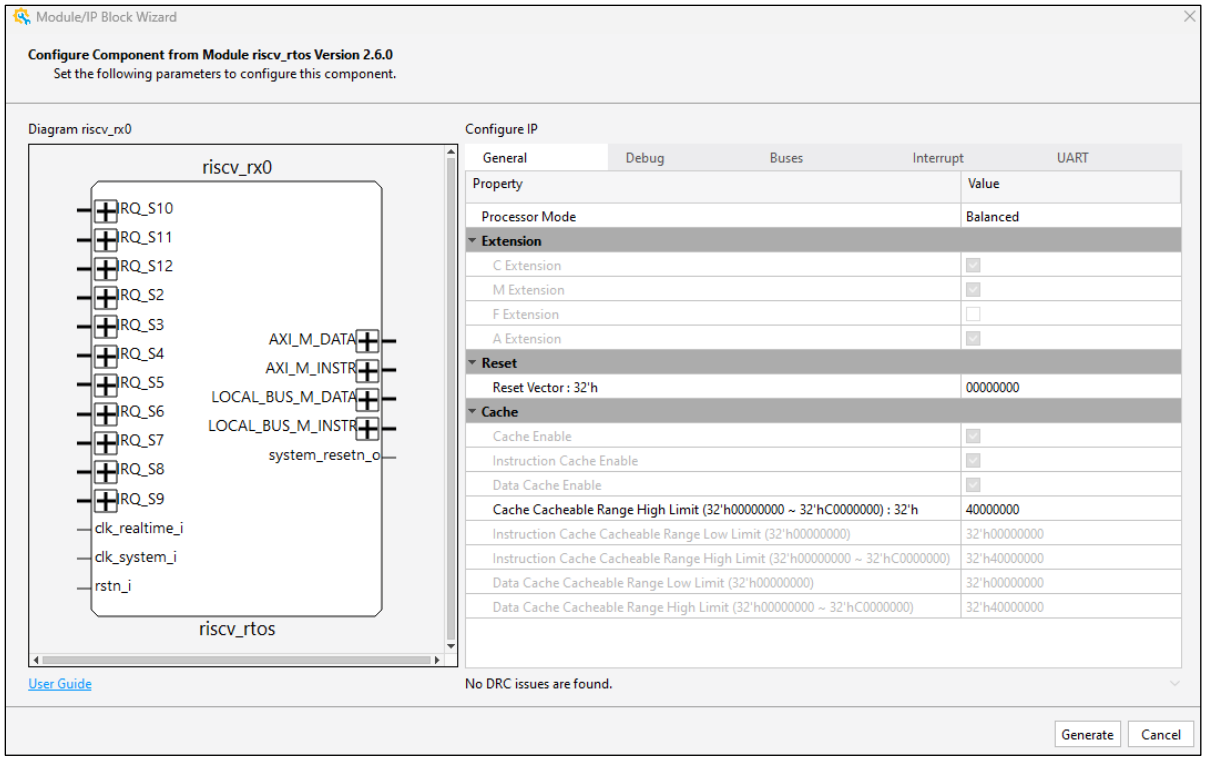

Figure 4.16. RISC-V RX CPU IP General Configuration

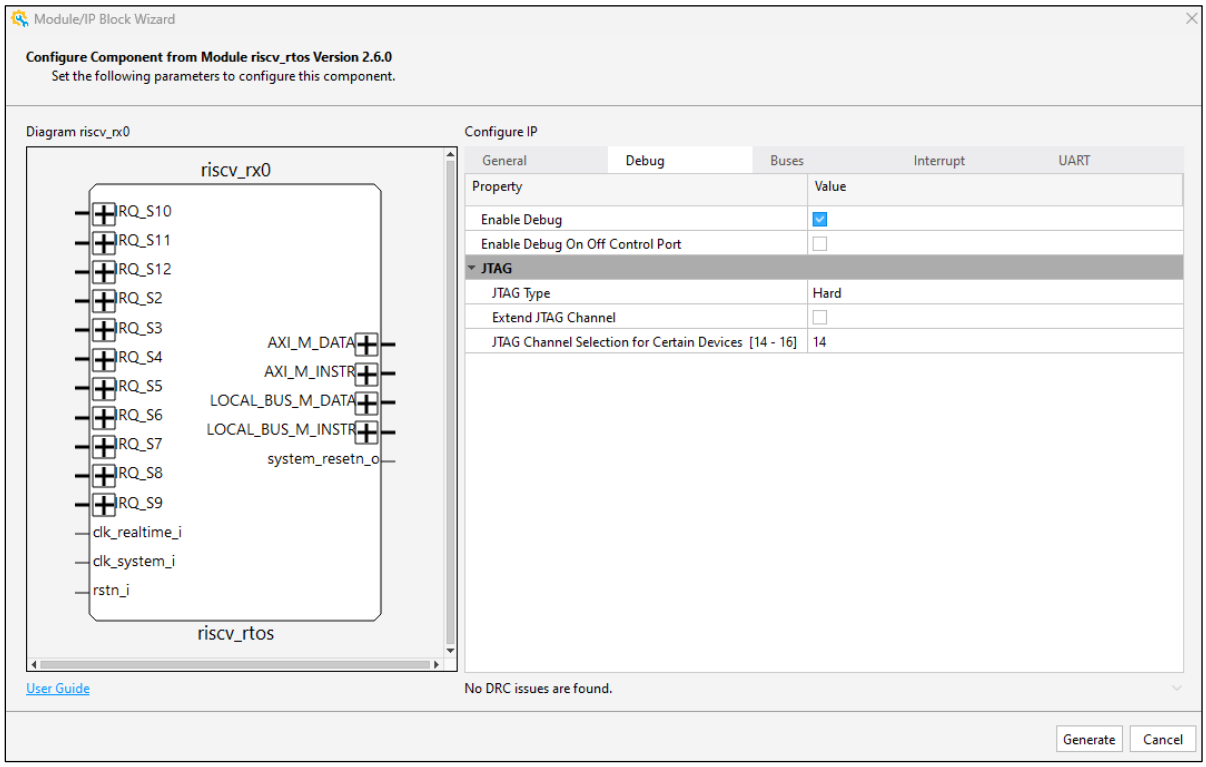

Figure 4.17. RISC-V RX CPU IP Debug Configuration

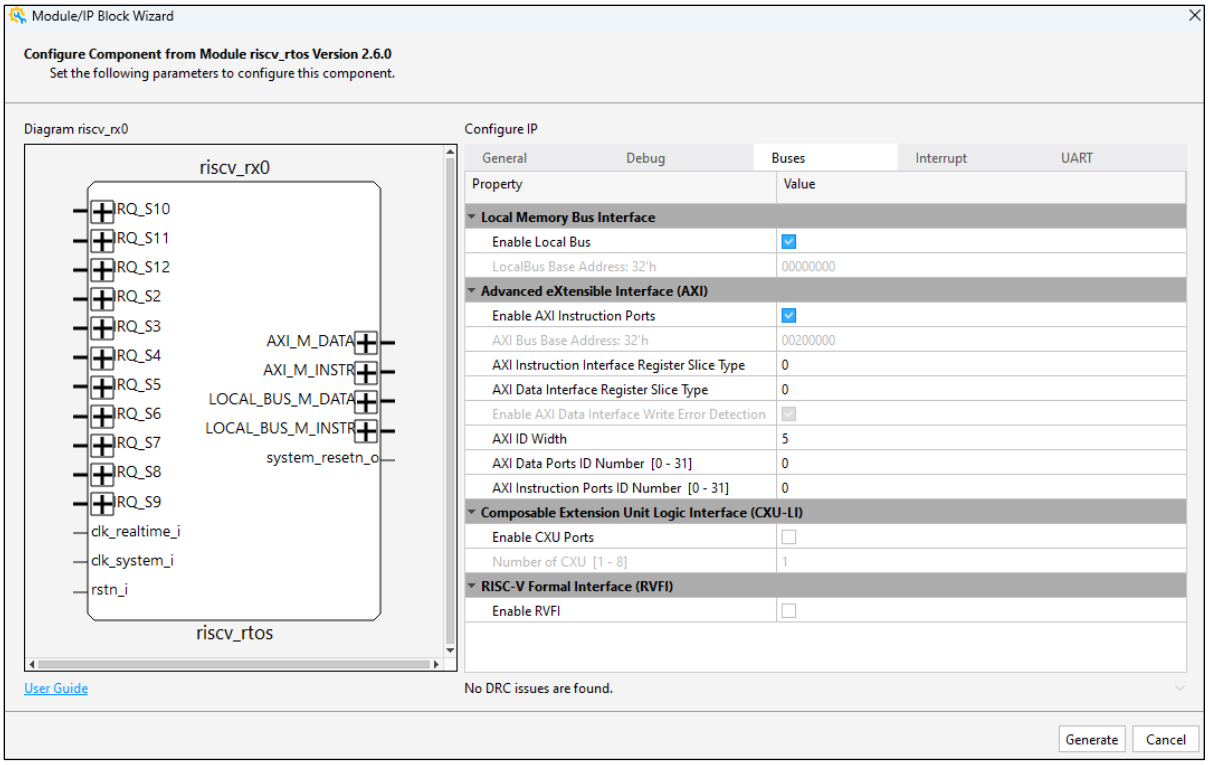

Figure 4.18. RISC-V RX CPU IP Buses Configuration

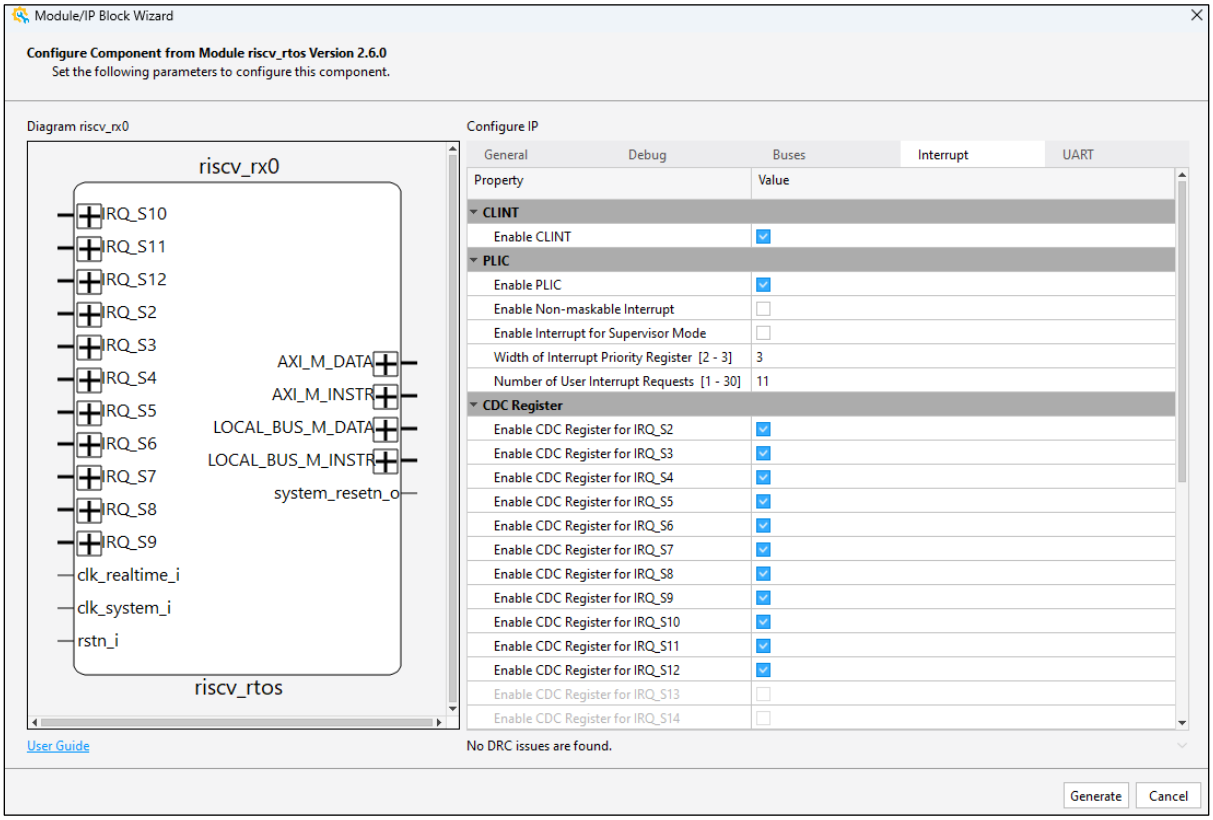

Figure 4.19. RISC-V RX CPU IP Interrupt Configuration

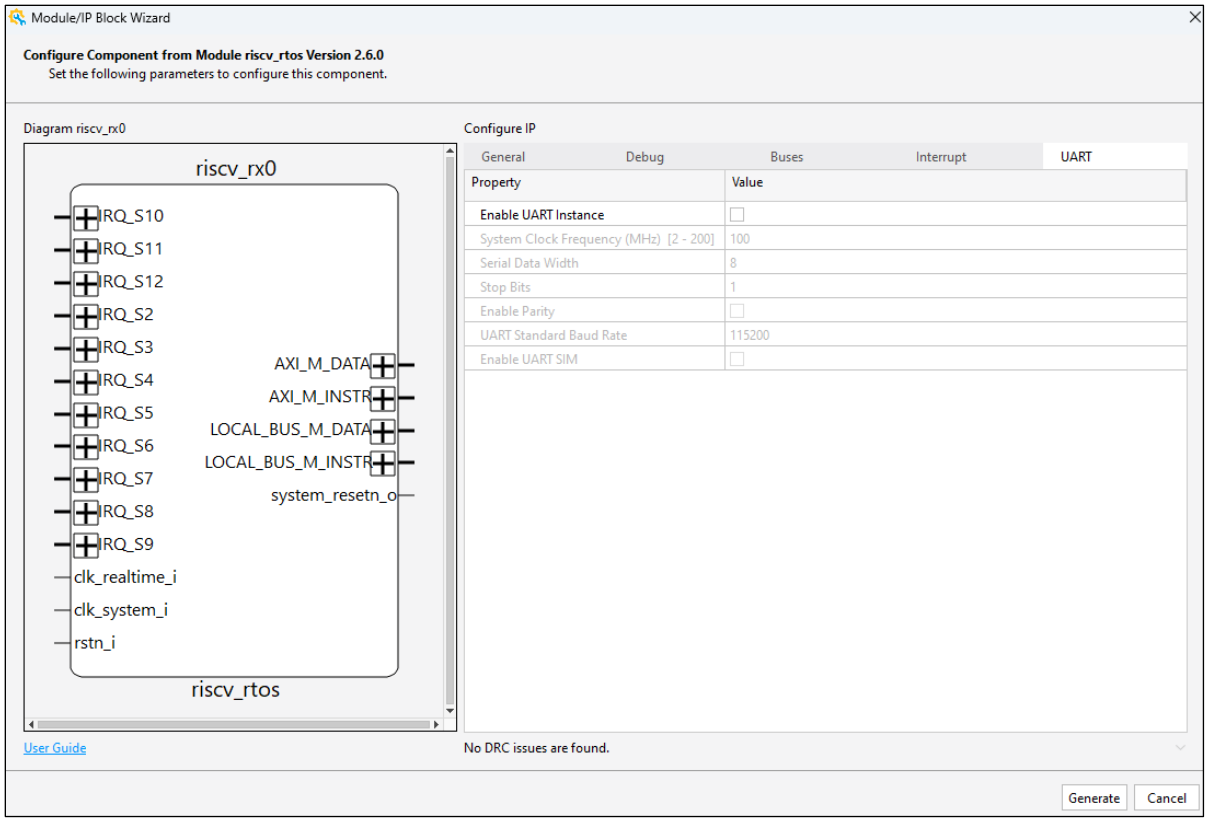

Figure 4.20. RISC-V RX CPU IP UART Configuration

4.14.3. ML IP

The Advanced CNN Accelerator IP is a calculation engine for deep neural networks with fixed-point weight. The IP calculates full layers of the neural network, including convolution layer, pooling layer, batch normalization layer, and fully connected layer, by executing a sequence of firmware code with weight values generated by the Lattice sensAI Neural Network Compiler. In the current design, the IP is configured with ML LRAM = 4 and VE SPD = 2.

For more information about the IP, refer to the Advanced CNN Accelerator IP User Guide (FPGA-IPUG-02224).

The following figures show the parameters selected and configured during IP instantiation in the design.

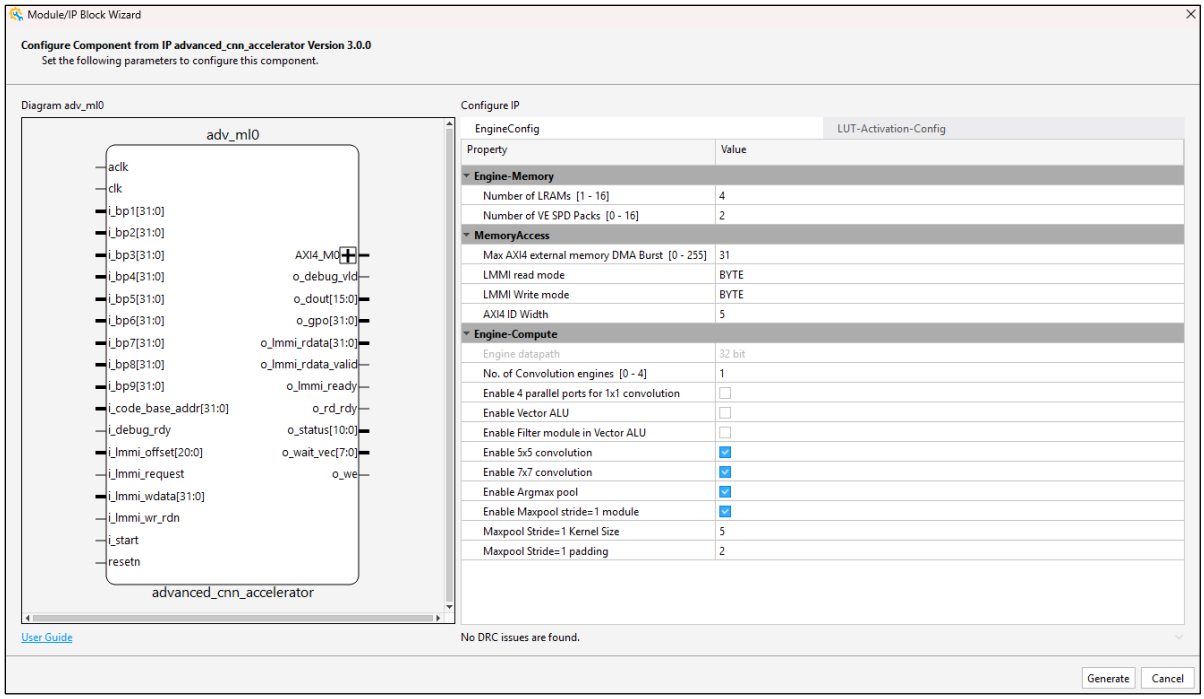

Figure 4.21. Advanced CNN Accelerator IP Engine Configuration

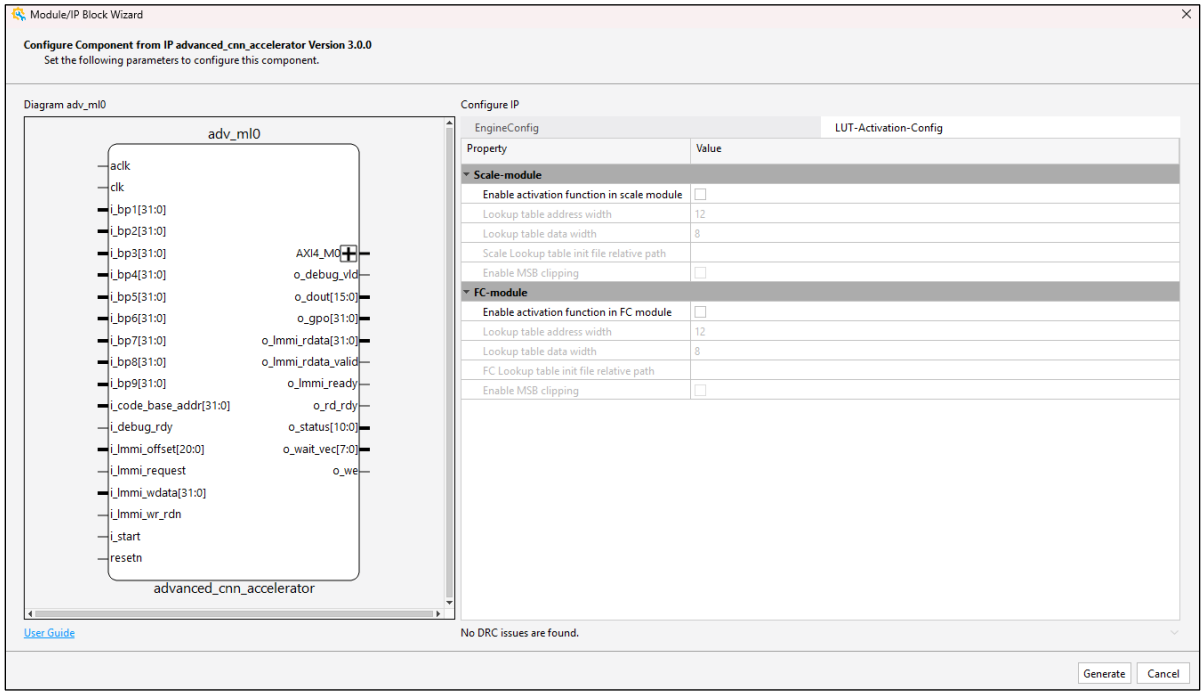

Figure 4.22. Advanced CNN Accelerator IP LUT-Activation Configuration

4.14.4. HyperRAM Controller

The controller sits between the AXI bus and two HyperRAM chips. The IP receives read and write requests from the AXI4 bus through the AXI4 subordinate interface and transfers data to the two HyperRAM chips on the FPGA board in parallel.

The following figure shows the parameters selected and configured during IP instantiation in the design.

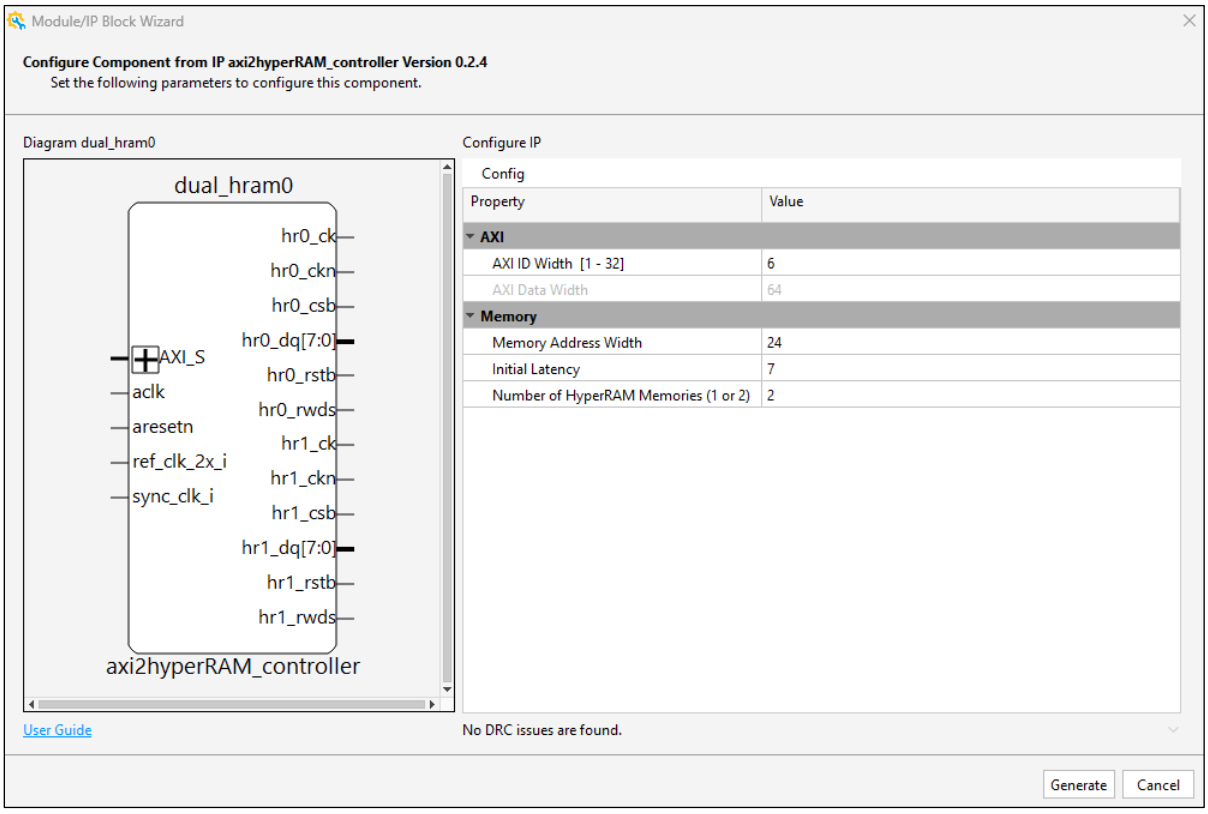

Figure 4.23. HyperRAM Controller IP Configuration

4.14.5. Octal SPI Controller

The Octal SPI is an eight tri-state data line serial interface that is commonly used to store, program, erase, and read SPI flash memories. Octal SPI enhances the throughput of a standard SPI by eight times because eight bits are transferred every cycle.

For more information about the IP, refer to the Octal SPI Controller IP User Guide (FPGA-IPUG-02273).

The following figures show the parameters selected and configured during IP instantiation in the design.

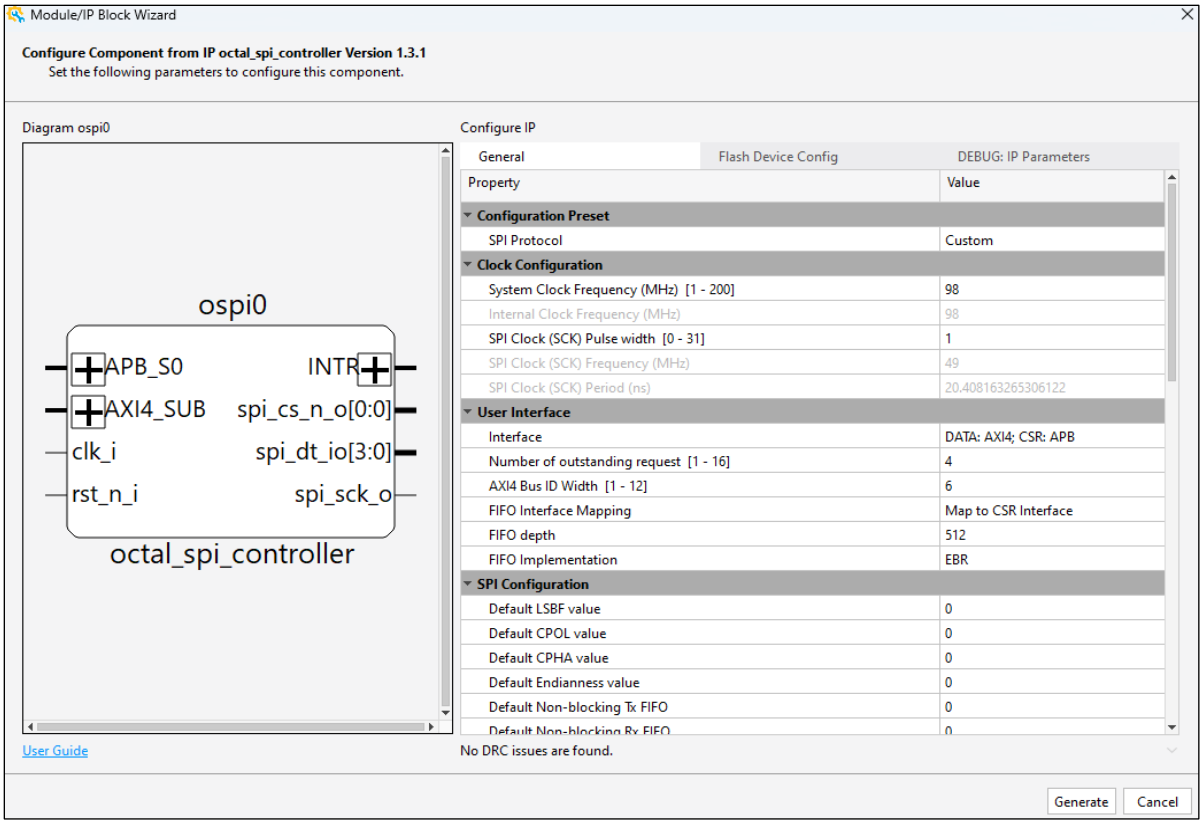

Figure 4.24. Octal SPI IP General Configuration 1

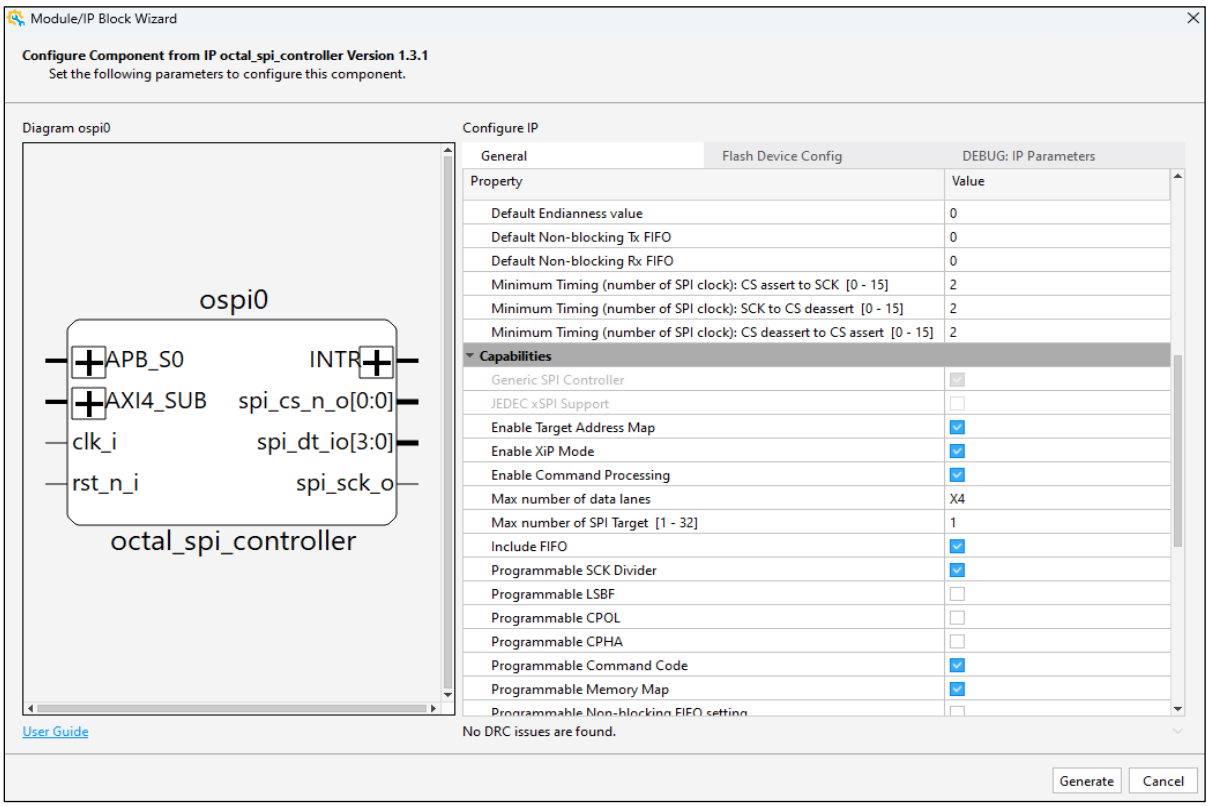

Figure 4.25. Octal SPI IP General Configuration 2

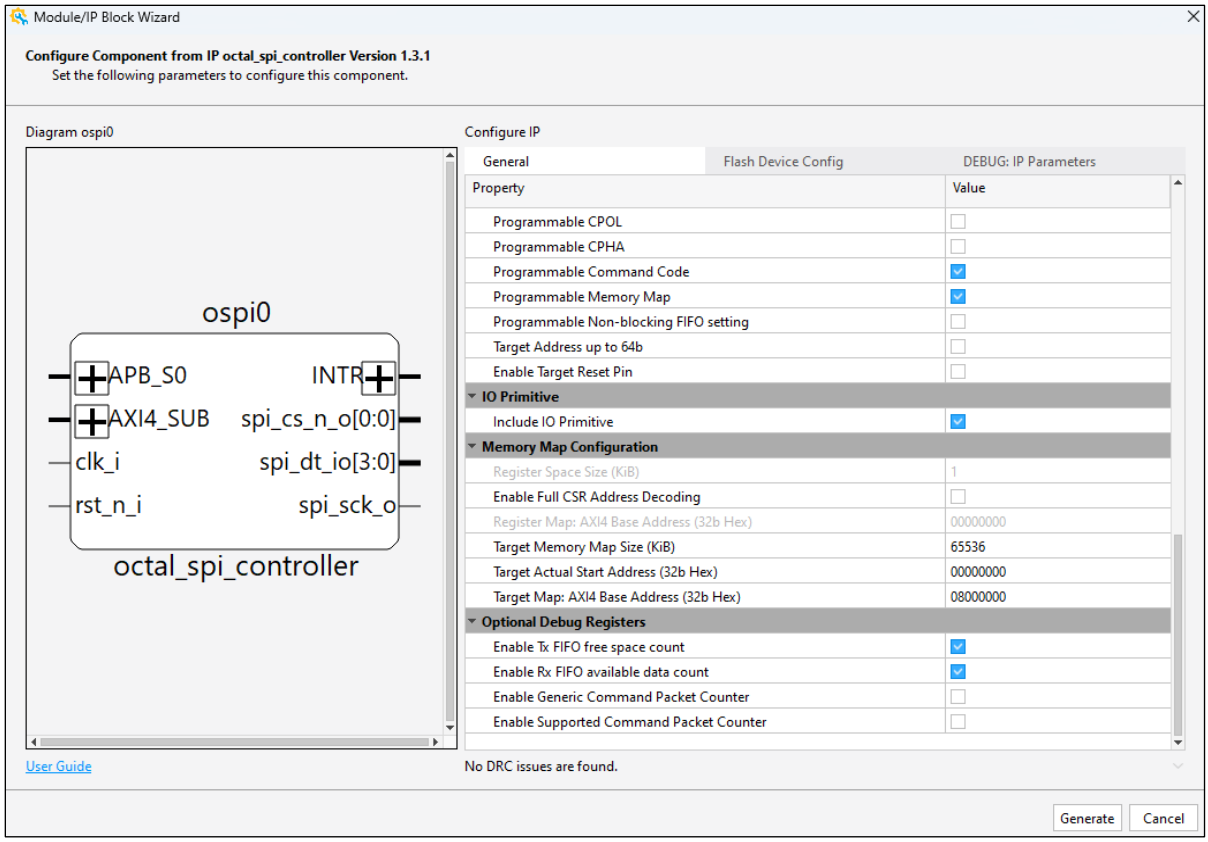

Figure 4.26. Octal SPI IP General Configuration 3

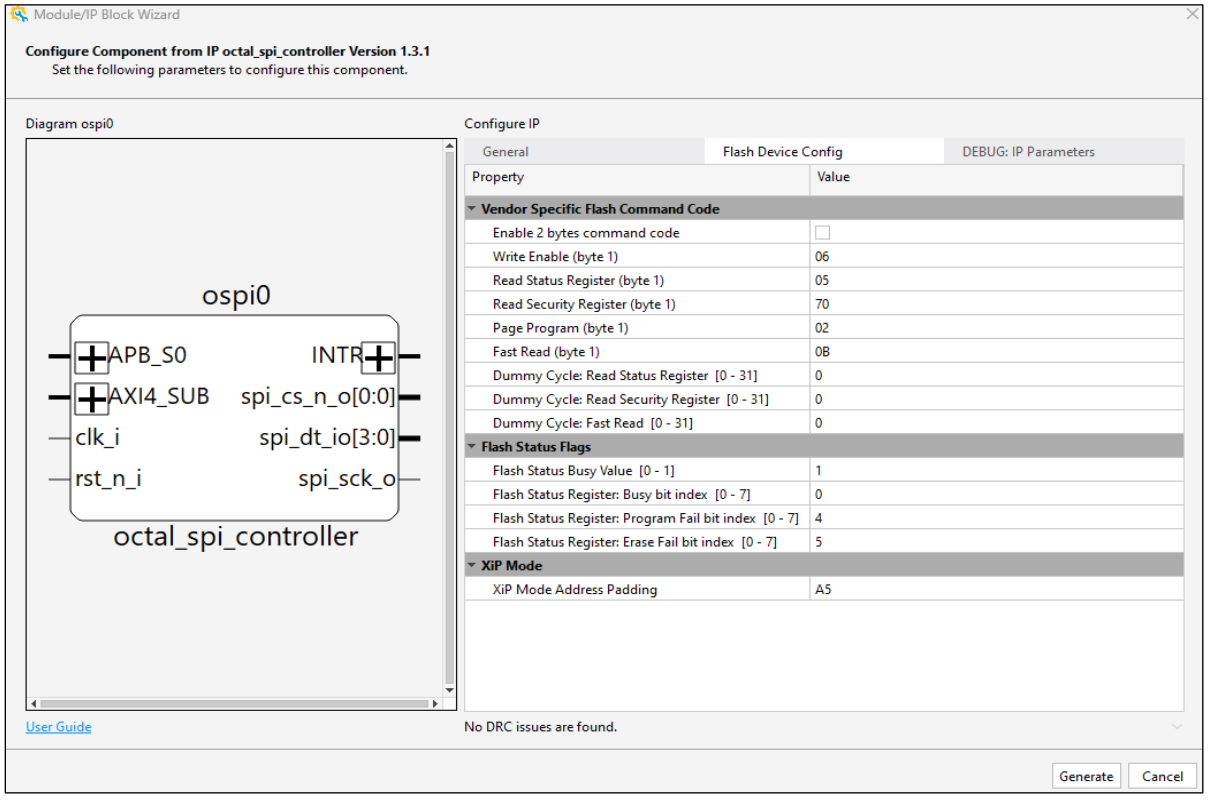

Figure 4.27. Octal SPI IP Flash Device Configuration

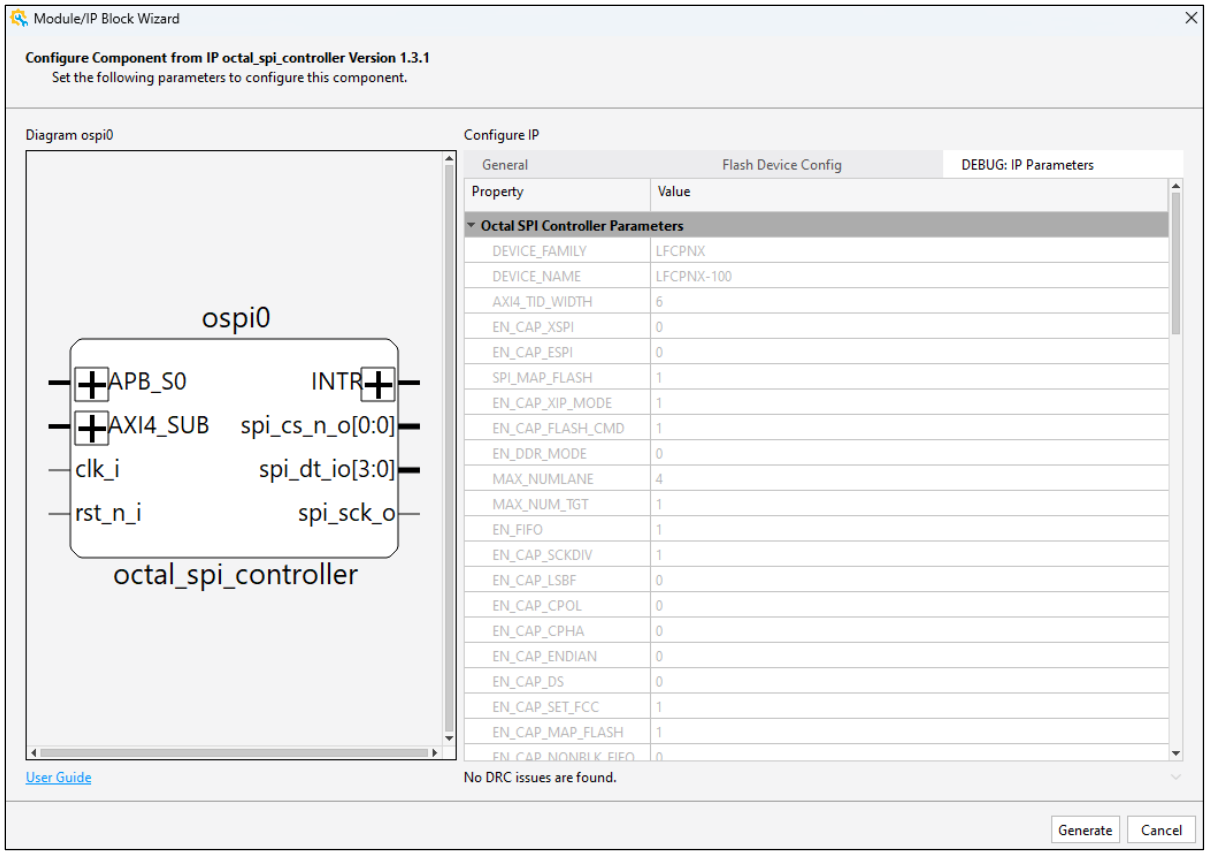

Figure 4.28. Octal SPI IP Debug IP Parameters Configuration

4.14.6. Tightly-Coupled Memory

The design implements 128 kB of Tightly-Coupled Memory (TCM). This IP is used for boot code and performance-critical routines.

The following figures show the parameters selected and configured during IP instantiation in the design.

For more information about the IP, refer to the Tightly-Coupled Memory IP User Guide (FPGA-IPUG-02299).

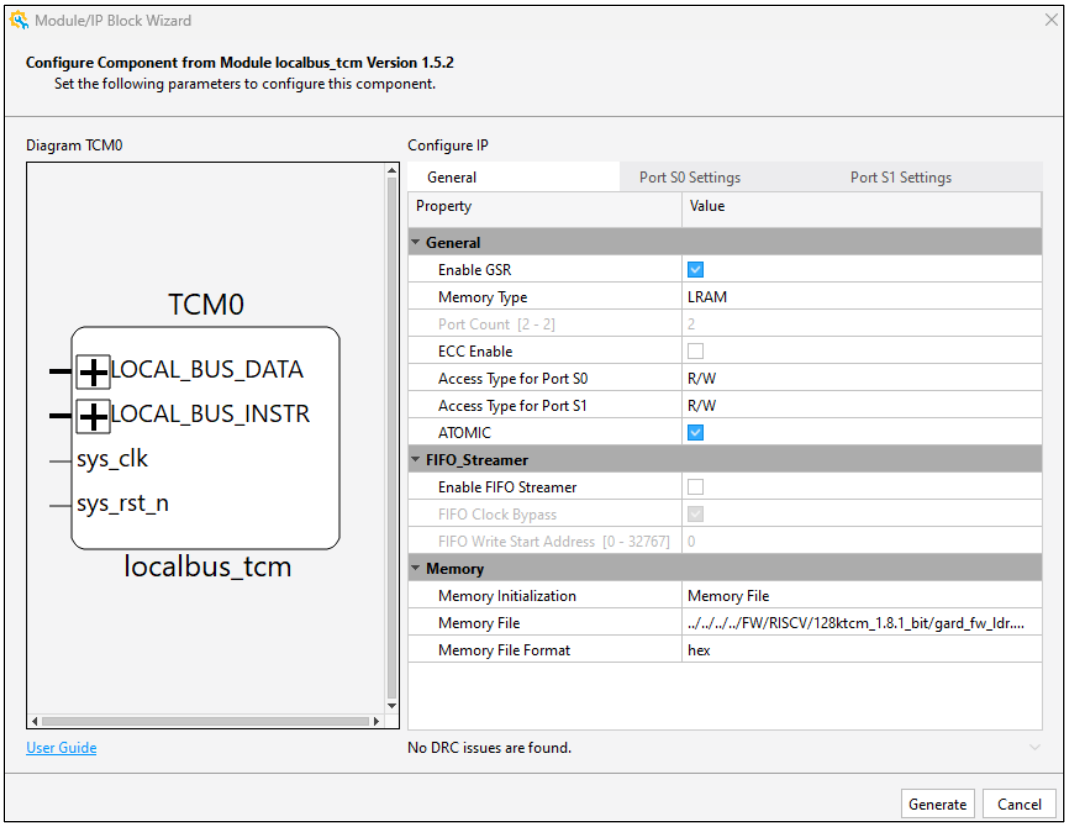

Figure 4.29. TCM IP General Configuration

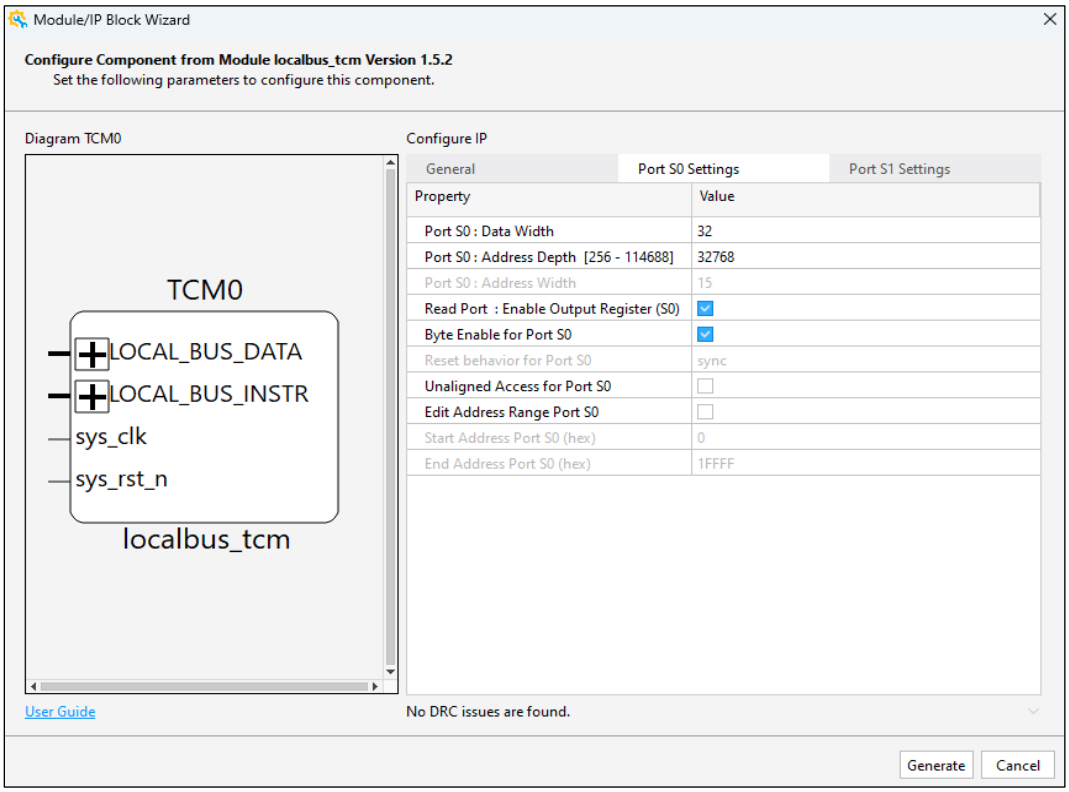

Figure 4.30. TCM IP Port S0 Settings

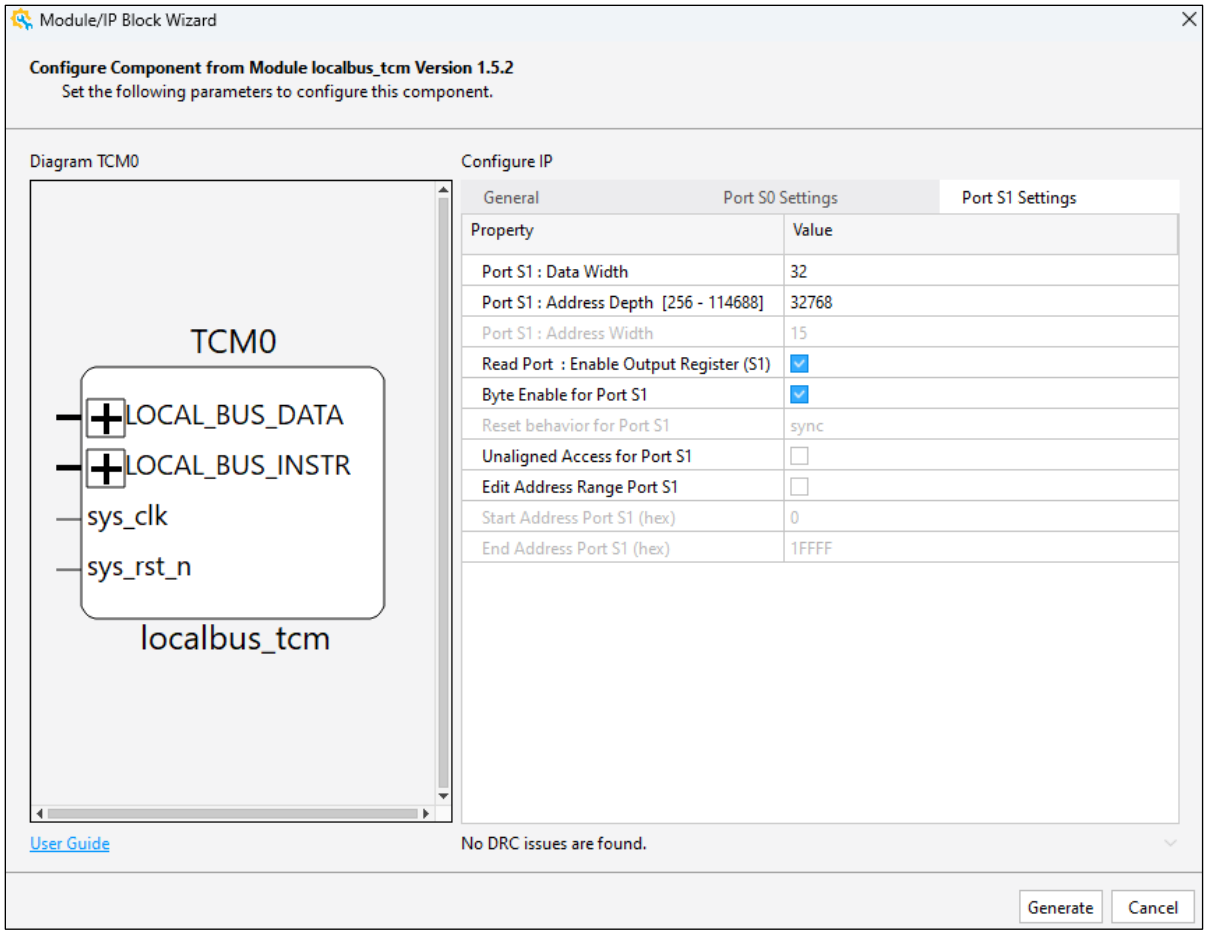

Figure 4.31. TCM IP Port S1 Settings

4.14.7. UART

The UART IP performs serial-to-parallel conversion of data characters received from a peripheral UART device and parallel-to-serial conversion of data characters received from the host locater insider the FPGA through an APB interface.

For more information about the IP, refer to the UART IP User Guide (FPGA-IPUG-02105).

The following figure shows the parameters selected and configured during IP instantiation in the design.

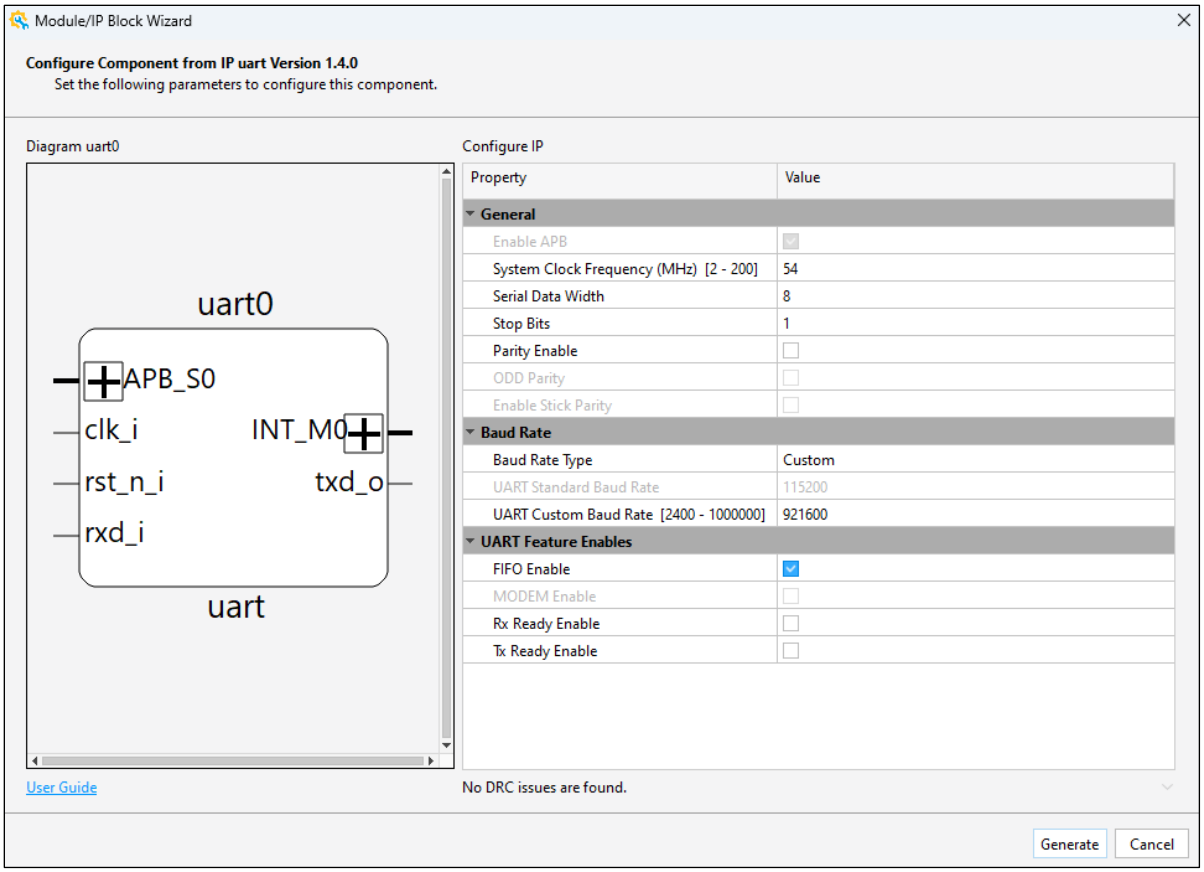

Figure 4.32. UART IP Configuration

4.14.8. I2C Controller

The I2C Controller IP provides a standard two-wire external I2C bus interface. This IP supports out-of-band interrupts and configurable transmit or receive FIFO size to minimize intervention by the host.

For more information about the IP, refer to the I2C Controller IP User Guide (FPGA-IPUG-02071).

The following figure shows the parameters selected and configured during IP instantiation in the design.

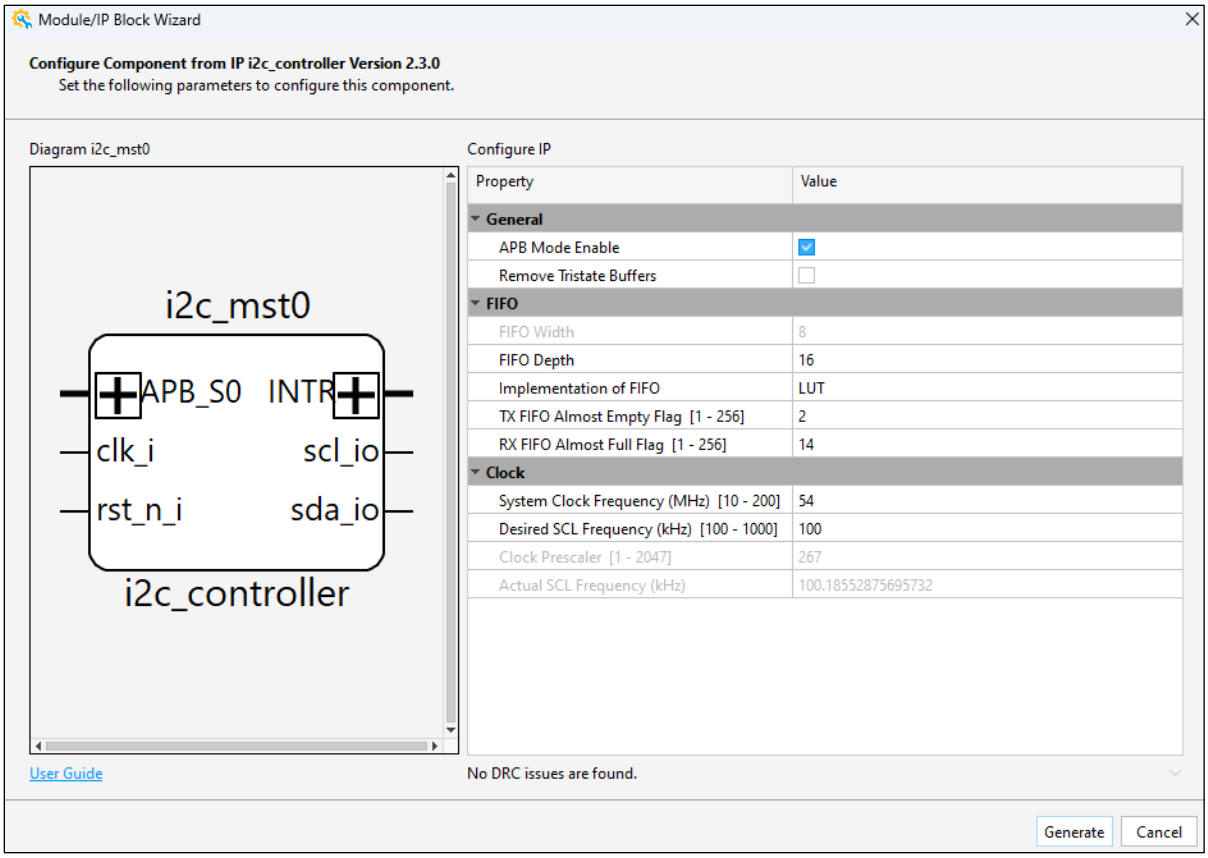

Figure 4.33. I2C Controller IP Configuration

4.14.9. I2C Target

The I2C Target IP provides a bridge between the LMMI/APB interface and the standard external I2C bus interface. This IP supports out-of-band interrupts as well as configurable transmit and receive FIFOs to minimize intervention by the host.

For more information about the IP, refer to the I2C Target IP User Guide (FPGA-IPUG-02072).

The following figure shows the parameters selected and configured during IP instantiation in the design.

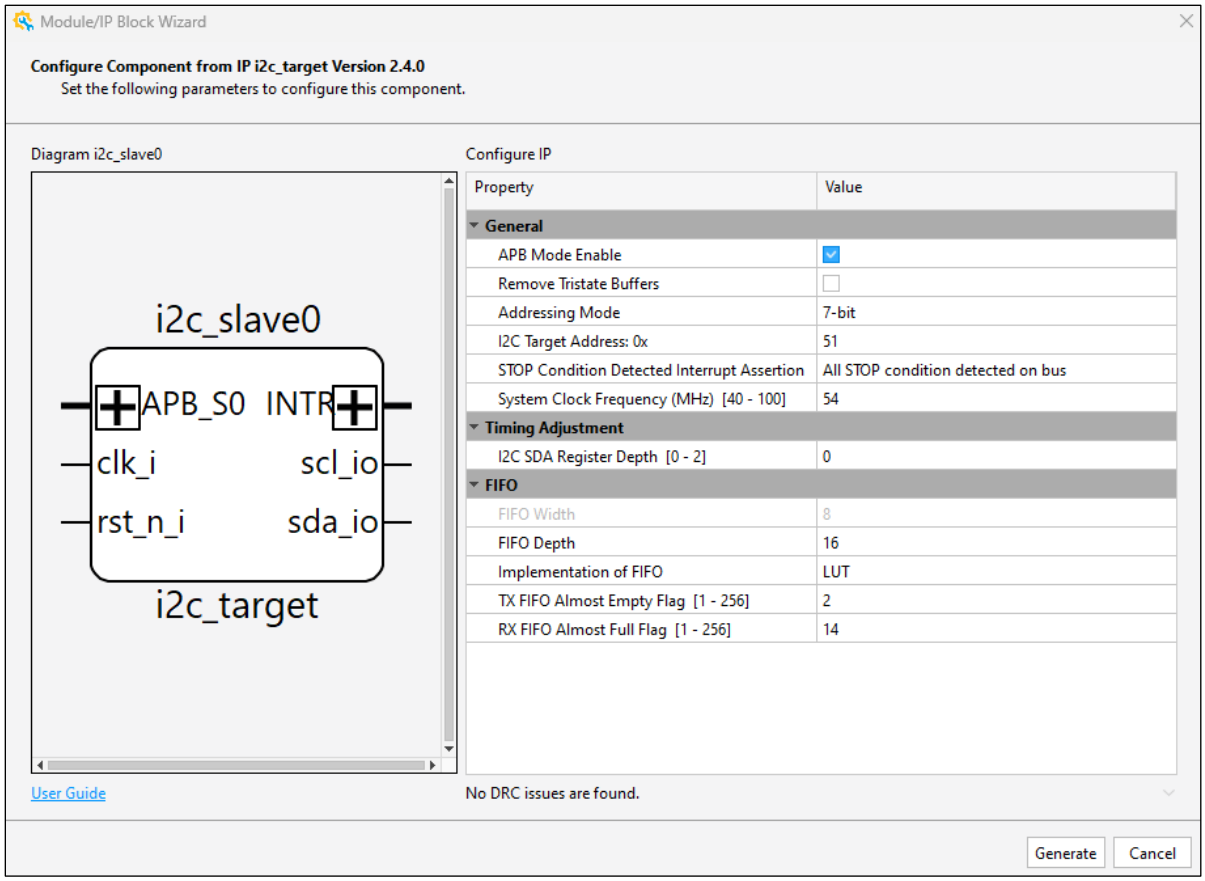

Figure 4.34. I2C Target IP Configuration

4.15. Customizing Reference Design

The GARD architecture combines Propel Builder-based design with conventional RTL using HDLs to integrate the top-level. Since the LFCPNX-100 GARD component is created using the Propel Builder tool, any modifications required in the component must be done in the Propel Builder tool. Modifications outside the LFCPNX-100 GARD component can be made in the respective RTL HDL files.

The following two examples demonstrate customization in the Propel Builder design and RTL HDL files.

4.15.1. Customizing I2C Target Device Address

The I2C target on the LFCPNX-100 GARD board uses a default address of 0x51. If you want to configure it to appear at a different address when accessed from a host device such as a Raspberry Pi, follow the steps below:

1. Launch the Lattice Propel Builder software.
2. Click **File** > **Open Design**, and from the project database, open the Propel Builder project file (*.sbx*) from the package.

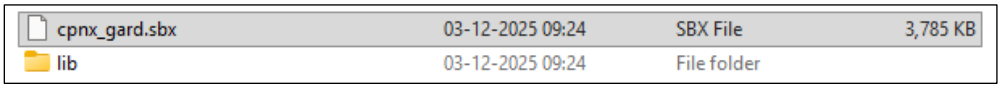

Figure 4.35. Open Propel Builder Project File

3. The opened project file looks as shown in the following figure.

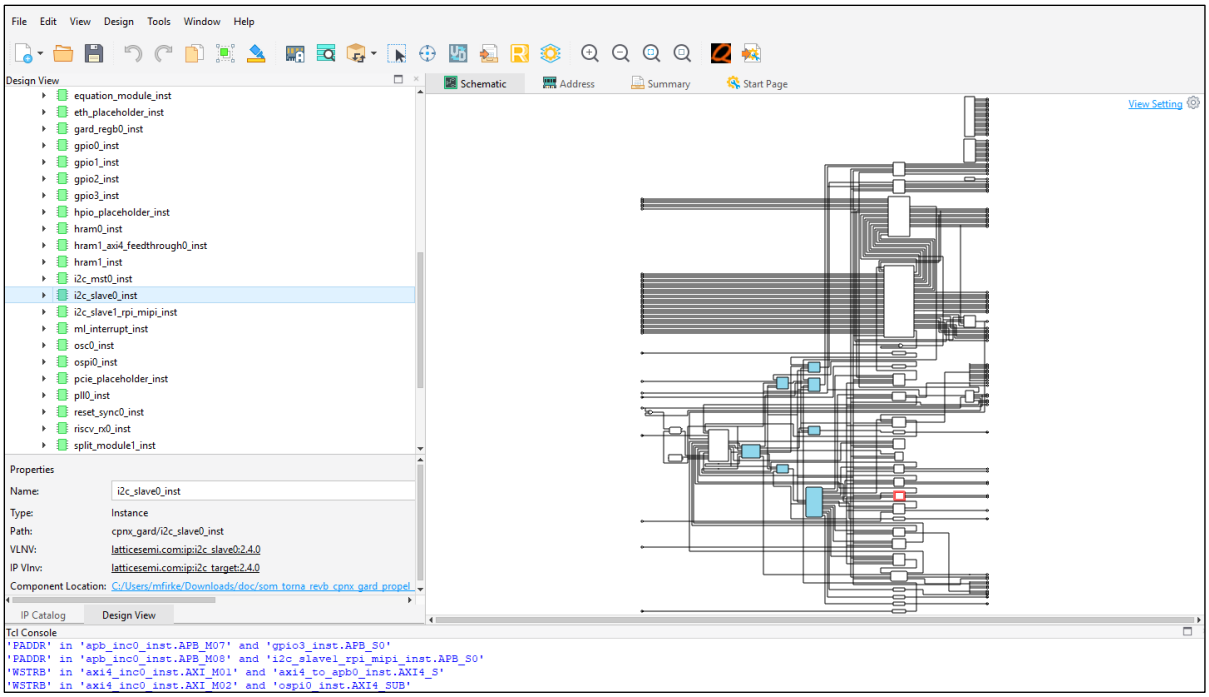

Figure 4.36. Opened Propel Builder Project File

4. In the design schematic, locate and zoom in to **i2c_slave0_inst**, and double-click this block.

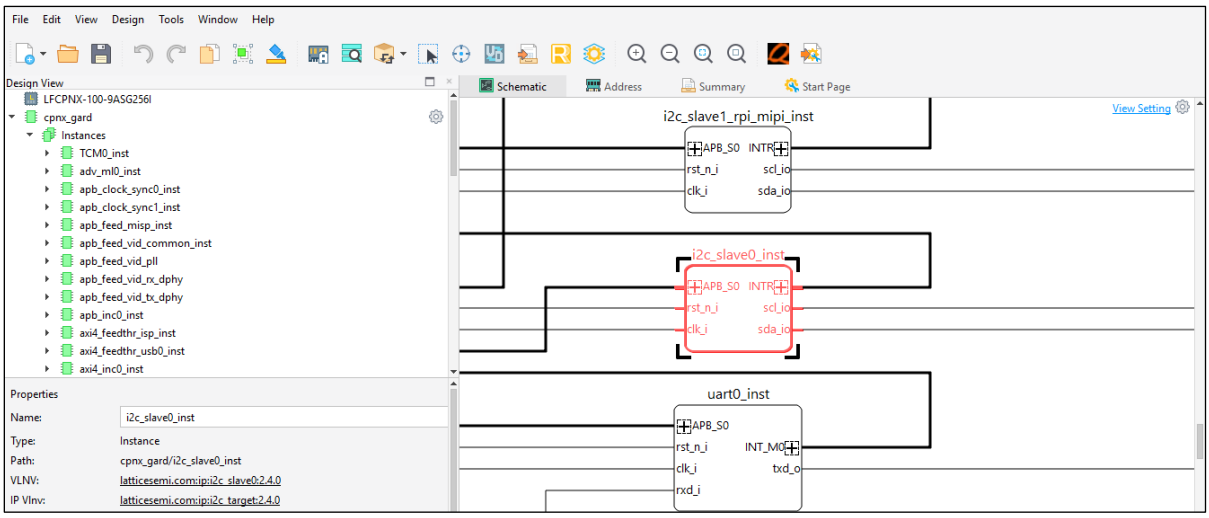

Figure 4.37. Zoom on I2C Target IP

5. In the IP configuration window, change the **I2C Target Address** to **0x27**, and click **Generate** at bottom right of the configuration window.

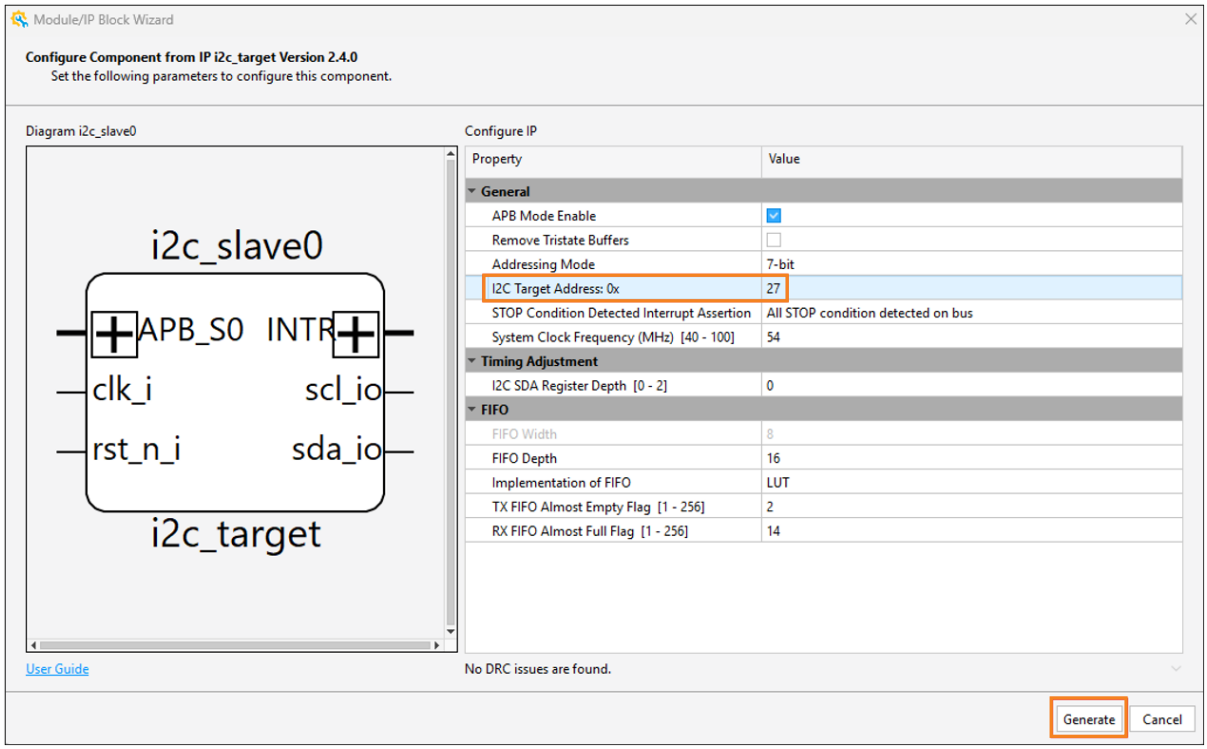

Figure 4.38. Configure I2C Target IP

6. Click **Finish** in the next pop-up window.
7. In the top tool button ribbon, click **Validate Design**.

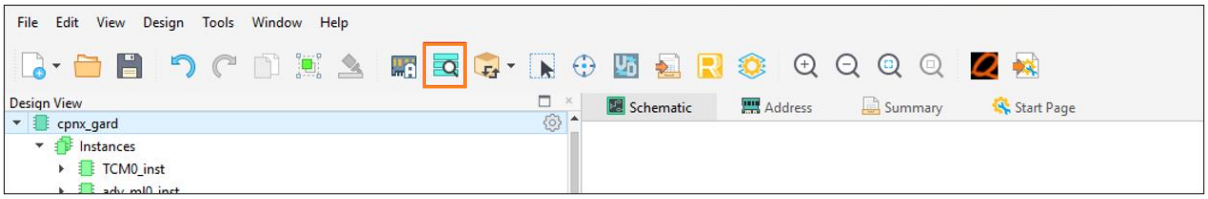

Figure 4.39. Validate Propel Builder Design

8. A completion message is displayed when the validation completes without any errors.

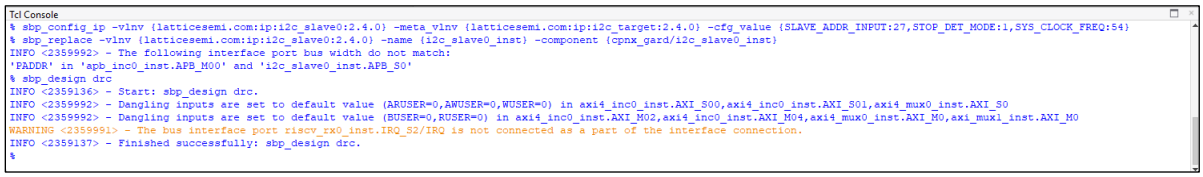

Figure 4.40. Validate Design Log

9. Click **Force Generate** so that all IPs and their connections are updated in the top level along with the new settings.

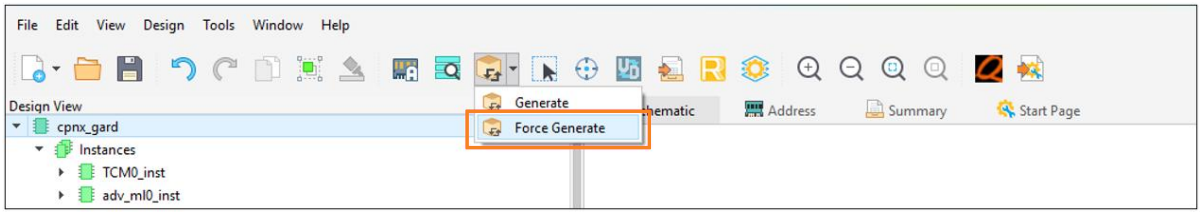

Figure 4.41. Force Generate

10. Click **Run Radiant** to launch the Radiant tool.

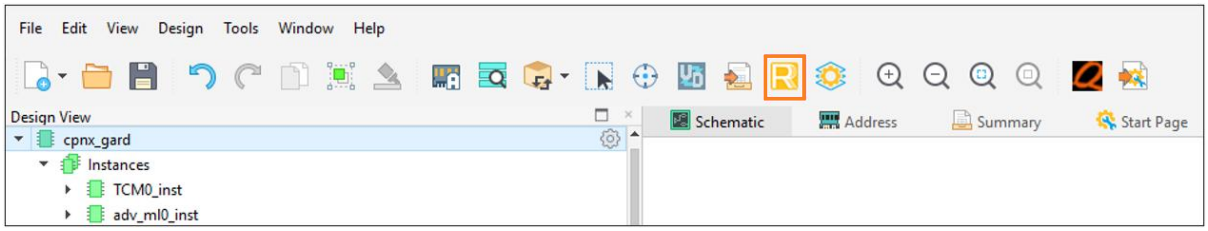

Figure 4.42. Run Radiant Software

11. The Lattice Radiant tool opens in a new pop-up window. In the Radiant window, click **File** > **Close Project**. Use the Radiant project provided in the package to run next steps of synthesis, as well as Place and Route.
12. Click **File** > **Open** > **Project** and open the Radiant project file (*.rdf*) from the folder: *mod_top_torna_revb_cpnx100_radiant/mod_top_torna_revb_cpnx100.rdf*.
Note: The current implementation of GARD RTL sources refers to *mod_**; however, these files are also applicable to other GARD RTL-based applications such as DD and HMI.

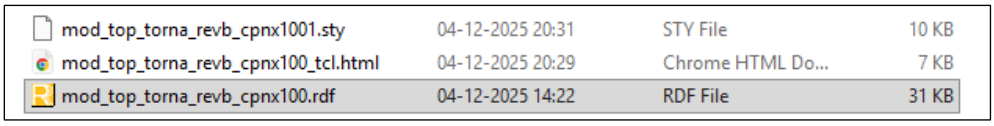

Figure 4.43. Open Radiant Project File

13. In the left-side file navigation window, under the **Input Files** folder, find the **i2c_slave0.ipx** file. Right-click the file and select **Remove**.

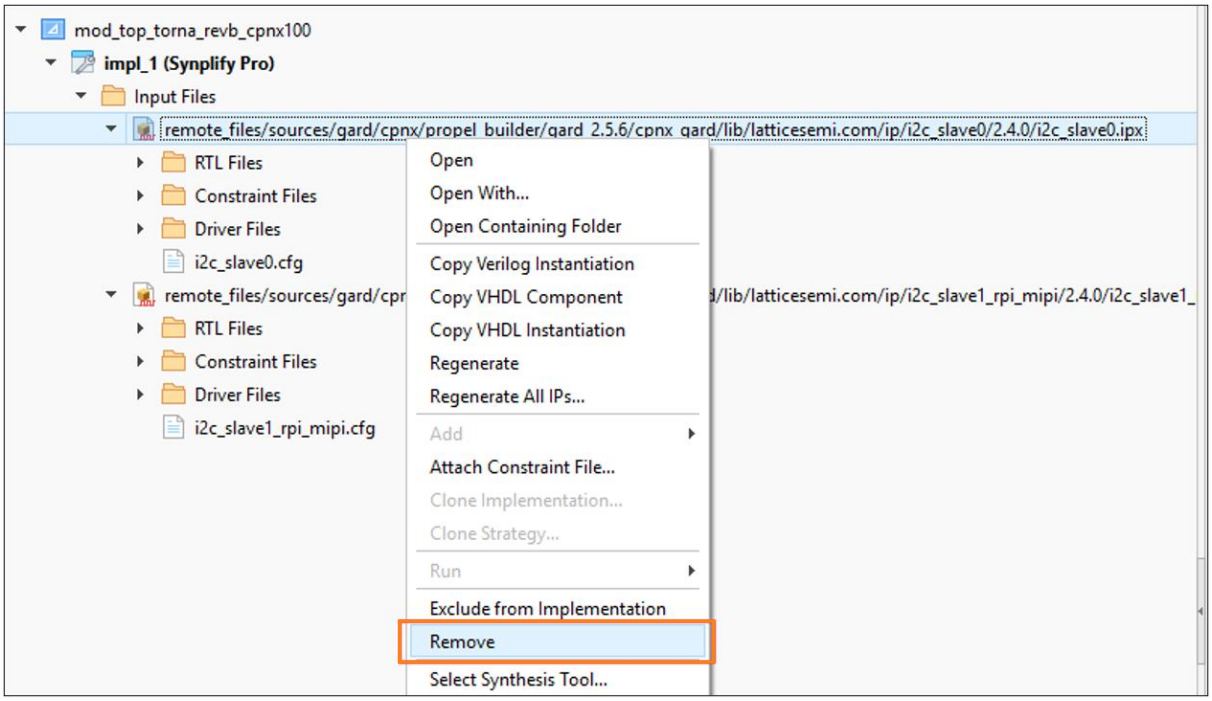

Figure 4.44. Remove Existing I2C Target IP

14. In the file navigation window on the left, right-click **Input Files** and select **Add** > **Existing File**.

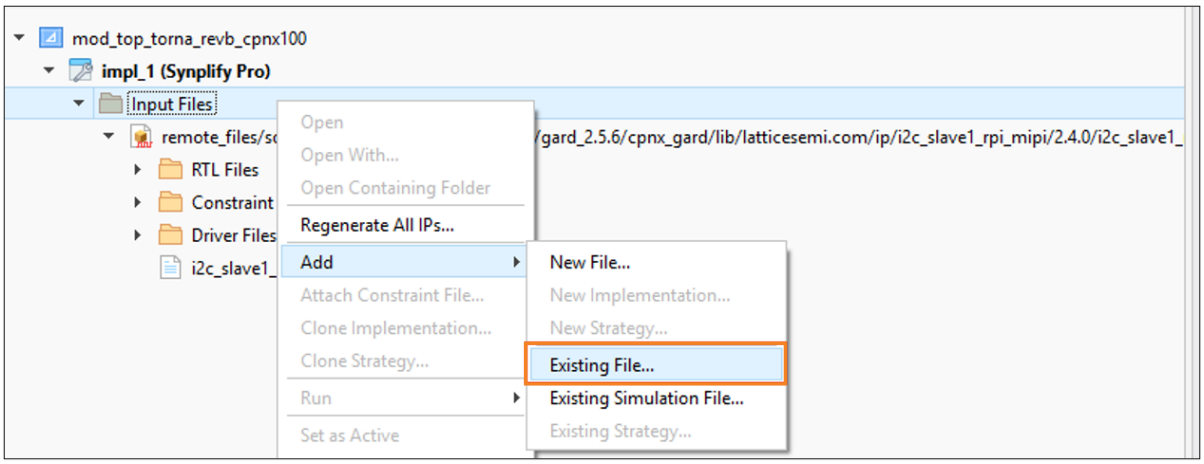

Figure 4.45. Add New I2C Target IP

15. Select the **i2c_slave0.ipx** file generated by the Propel Builder tool in Step 6 and click **Add**. The file is located in the folder:
som_torna_revb_cpnx_gard_propel_builder/som_torna_revb_cpnx_gard/cpnx_gard/lib/latticesemi.com/ip/i2c_slave0/2.4.0.
16. Once the above file is added, the design hierarchy reflects the new *.ipx* file as shown in the figure below.

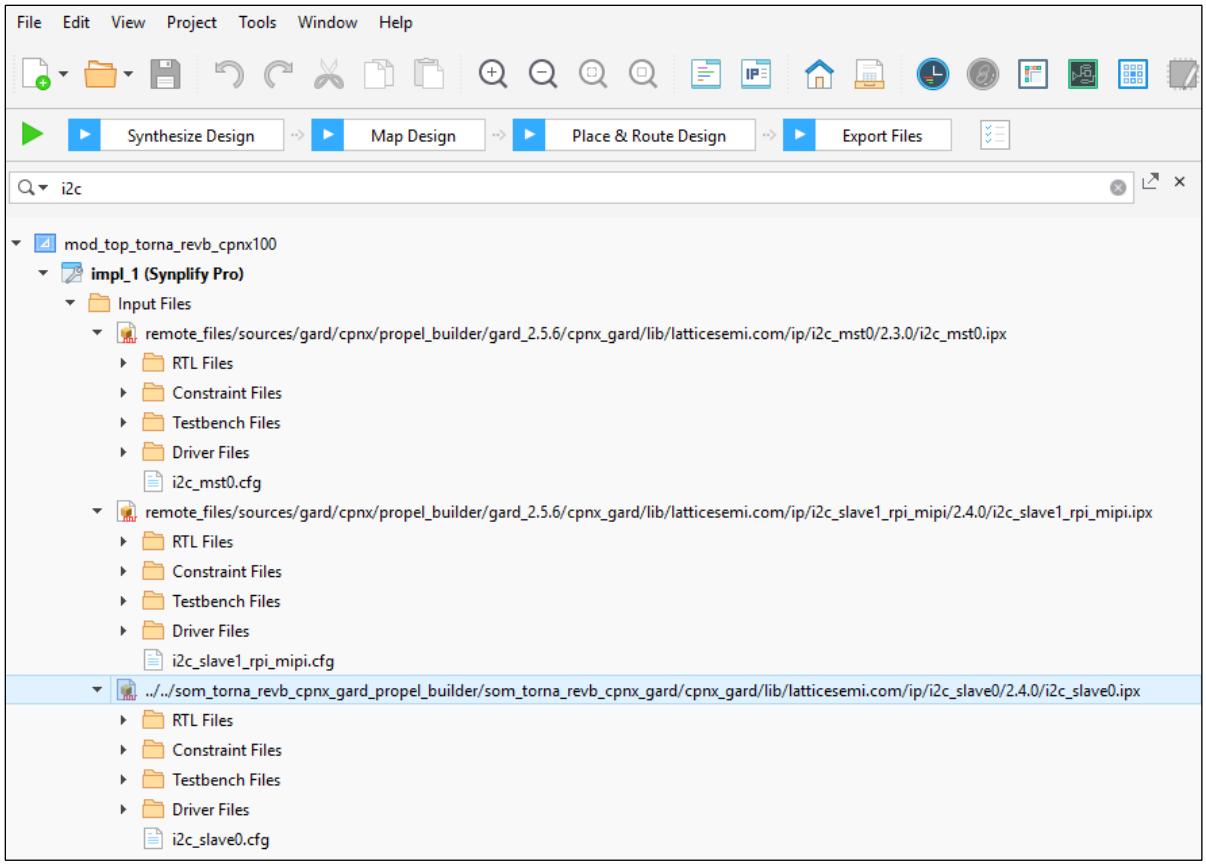

Figure 4.46. Modified Radiant Project Hierarchy

17. The Radiant project setup is now ready. Click **Export Files** to generate the bitstream file.

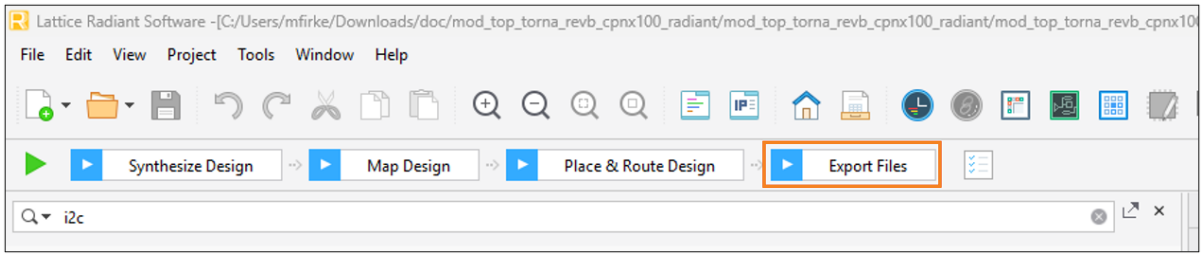

Figure 4.47. Export Files

18. View the log messages in the output window for the generated bitstream. Find the generated *.bit* file in the folder: *mod_top_torna_revb_cpnx100_radiant/impl_1*.

4.15.2. Generating Video Pipeline with MIPI 2-Lane Passthrough

The default GARD reference design configures the application subsystem for both input and output with 4 MIPI lanes at 3,280 × 2,464 resolution and 30 FPS. This example demonstrates the steps required to change both input and output of the application subsystem to 2 MIPI lanes at 1,920 × 1,080 resolution and 30 FPS:

1. Open the *cpnx_video_path_directives.vh* Verilog header file located in the folder: *mod_top_torna_revb_cpnx100_radiant/mod_top_torna_revb_cpnx100_radiant/remote_files/include*.
2. Edit this file as follows:
   - Uncomment the following defines:
     - MIPI_RX_2L_FIXED

    - MIPI_TX_2L_FIXED
    - FIXED_PLL_SETTING "2L_912MBPS"
    - INPUT_2TO1_DS_ALWAYS_OFF
  - Comment out the following defines:
    - MIPI_RX_4L_FIXED
    - MIPI_TX_4L_FIXED
    - FORCE_CLK_HS_IN_HB
    - INPUT_2TO1_DS_ALWAYS_ON
    - FIXED_PLL_SETTING "4L_720MBPS_PT_OR_DS_1L"
  - Change *RX_FIFO_DEPTH* to *64*.
3. Save the *cpnx_video_path_directives.vh* file.
4. Launch the Lattice Radiant software.
5. Click **File** > **Open Project**, and from the project database, open the Radiant project file (*.rdf*) from the folder: *mod_top_torna_revb_cpnx100_radiant/mod_top_torna_revb_cpnx100.rdf*.

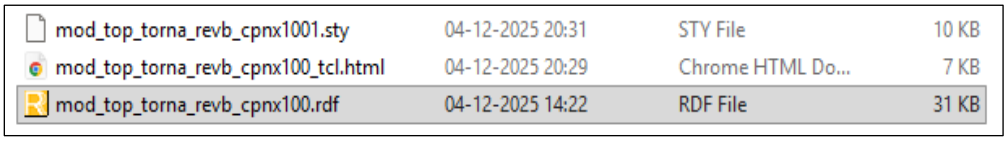

Figure 4.48. Open Radiant Project File

6. In the left-side file navigation window, under the **Input Files** folder, navigate to the **Post-Synthesis Constraint Files** folder. Right-click the folder and select **Add** > **Existing File**.

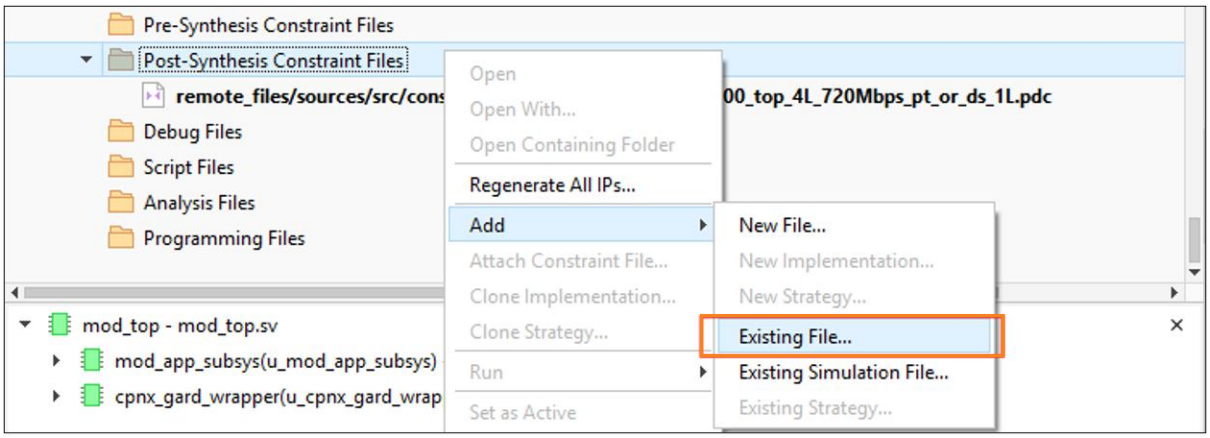

Figure 4.49. Add Constraint File

7. Select the *mod_torna_revb_cpnx100_top_2L_912Mbps.pdc* file from the folder: *mod_top_torna_revb_cpnx100_radiant/remote_files/sources/src/constraints*.
8. Click **Add**.
9. Right-click the newly added *.pdc* file and select **Set as Active**.

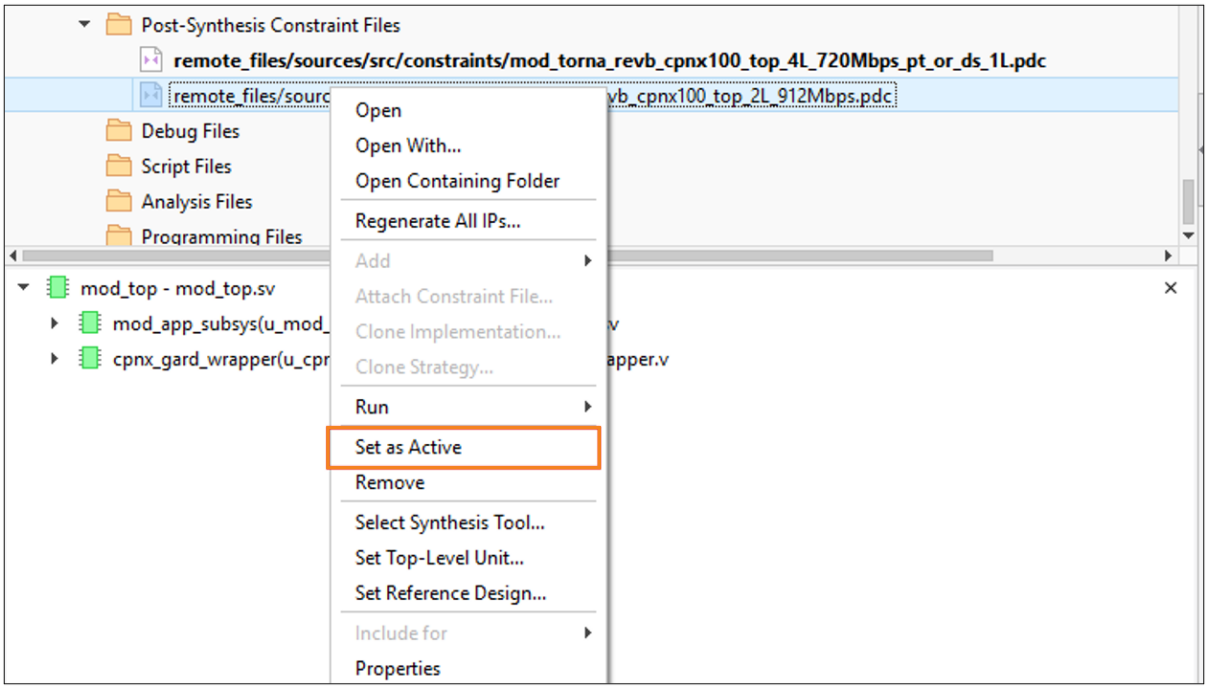

Figure 4.50. Set New Constraint File as Active

10. The Radiant project setup is now ready. Click **Export Files** to generate the bitstream file.

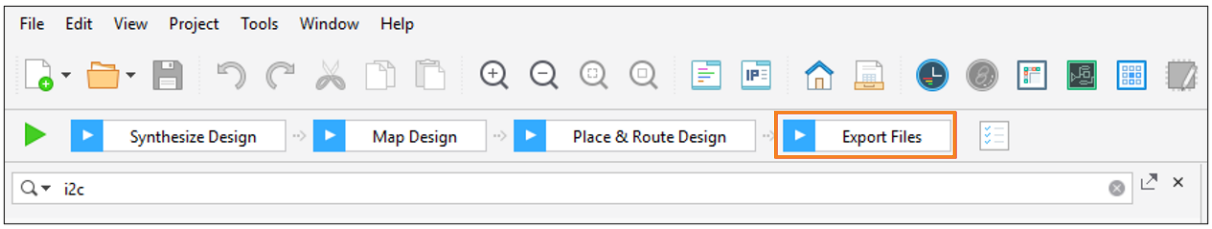

Figure 4.51. Export Files

11. View the log messages in the output window for the generated bitstream. Find the generated *.bit* file in folder: *mod_top_torna_revb_cpnx100_radiant/impl_1*.

5. GARD Firmware

5.1. Overview

GARD Firmware runs on RISC-V and executes tasks such as orchestrating system initialization, resource management, and application execution. This section explains its layered architecture and how this architecture enables quick and hardware-independent development of GARD-based application firmware modules such as MOD, HMI, and DD.

GARD Firmware consists of the following components:

- GARD Firmware Loader (Firmware Loader) – Loads the GARD Firmware binary that contains the Platform Firmware and Application Firmware linked together.
- GARD Platform Firmware (Platform Firmware) – Is GARD-aware and manages GARD resources to maximize power and performance based on available resources and user requirements.
- GARD Application Firmware (Application Firmware) – Uses helper routines from GARD Platform Firmware to achieve application-specific tasks.

The following figure shows an overview of GARD hardware and GARD firmware components.

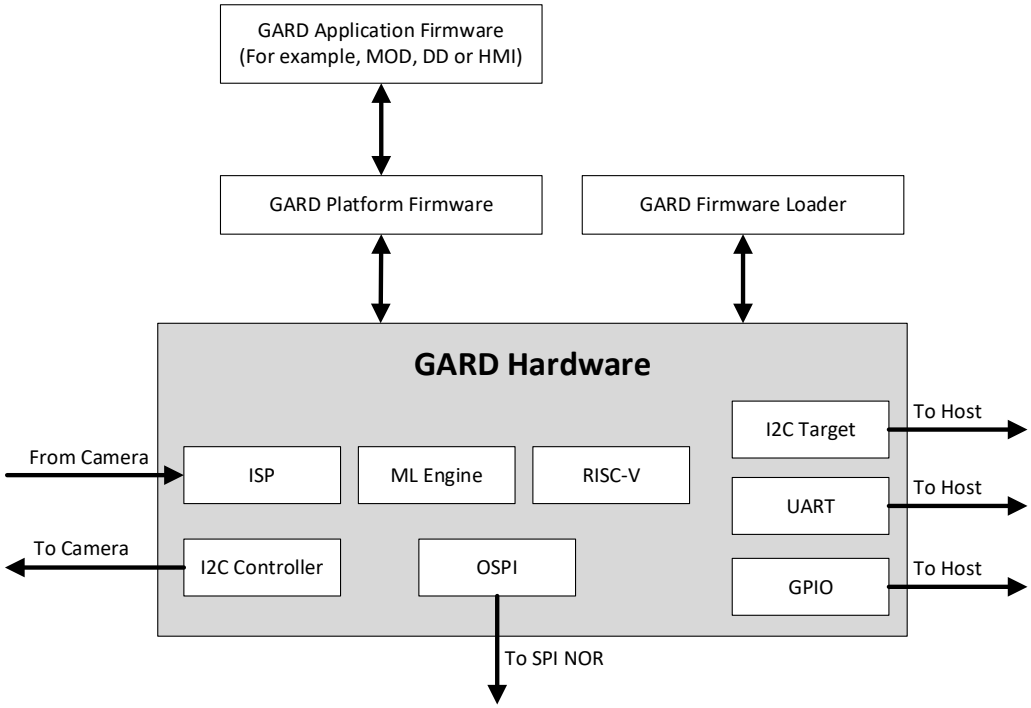

Figure 5.1. Layered Stack of GARD Hardware and GARD Firmware Components

The GARD hardware is located at the bottom of the stack. Running on RISC-V and controlling the GARD hardware is the GARD Platform Firmware. Stacked above the GARD Platform Firmware is the GARD Application Firmware, which uses the well-defined API (Application Programming Interface) of the GARD Platform Firmware to carry out its application tasks. This distinction enables independent and parallel code development, reducing delivery times. Additionally, the API ensures that updates to any one module do not obsolete the others.

The following sections document the GARD Platform Firmware and GARD Application Firmware modules. On one side is the Firmware Loader, which uses GARD hardware resources to locate, load, and execute firmware.

5.2. Firmware Loader

When the RISC-V contained within the GARD hardware is taken out of reset, the RISC-V starts executing the Firmware Loader code. The tasks of the Firmware Loader are:

- Perform any necessary POST.
- Locate and load firmware present in RFS.
- Support running any diagnostic firmware, if instructed.

Once the Firmware Loader has loaded the firmware, the Loader ceases to exist because the space where it operated in TCM is now occupied by the firmware.

5.2.1. Components

The Firmware Loader consists of the RFS Parser and the Loader.

5.2.1.1. RFS Parser

The RFS Parser reads the RFS configuration and uses this information to parse the RFS directory for locating the firmware that needs to execute on the GARD hardware. Since the Firmware Loader is not expected to be modified as frequently as the firmware, the RFS structure is designed carefully so that new revisions do not break the RFS Parser, which is part of the Firmware Loader.

5.2.1.2. Loader

After the RFS Parser routines have located the firmware in the RFS space, the Loader reads the firmware code from the SPI Flash and writes the code to TCM.

5.2.2. Operation

The following figure shows the typical flow of the Firmware Loader in non-error cases.

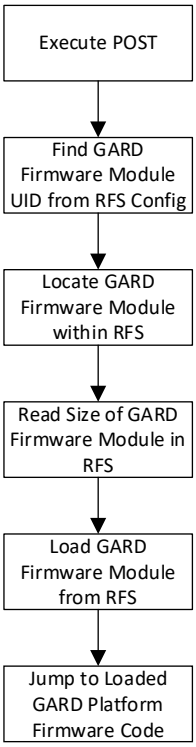

Figure 5.2. Firmware Loader Execution Flow

During normal operation, the Firmware Loader performs these steps:

- Execute POST – On bootup, the Firmware Loader initializes its environment and performs basic POST to verify that peripherals required for loading firmware are operational.
- Read RFS configuration – Access the RFS configuration space to determine the UID of the module containing the firmware.
- Locate firmware module and read its size – Identify the firmware module and determine its flash address and size.
- Load firmware into TCM – Read the firmware from flash using RFS routines and write the firmware to TCM.
- Execute firmware – After populating TCM with the firmware code, the Firmware Loader executes the firmware code.
- Support multi-phase loading (if required, in cases where TCM cannot hold the complete firmware):
  - Use the Phase 1 size (less than the complete firmware size) embedded in the firmware header.
  - Load only the Firmware code up to the size mentioned in the header.
  - Allow the firmware to load remaining phases into memory during execution.

5.3. Platform Firmware

Once the Firmware Loader completes its task, control passes to the Platform Firmware. Platform Firmware provides foundational services for Application Firmware. The Platform Firmware is tuned for the underlying GARD hardware and provides a stable API for GARD Application Firmware, abstracting it from hardware-specific details. The following sections document the features and responsibilities of the GARD Platform Firmware.

5.3.1. Features

Platform Firmware has the following features:

- Minimum execution overhead – Achieved by implementing the firmware as a bare-metal loop with low memory requirements.
- Various host communication interfaces – Includes UART and I2C.
- Data persistency – Maintains module data in RFS, which resides in SPI Flash.
- Multiple code execution memory modes – Supports TCM, XIP (execute-in-place from SPI Flash), and HyperRAM to meet different memory requirements.
- Support for Various Camera Vendors and Models – Enables flexibility through Camera Configuration extensions without modifying firmware code.

5.3.2. Components

The Platform Firmware consists of several components that provide essential services for managing memory, communication, and hardware resources. These components abstract the complexity of GARD hardware and offer a consistent interface for Application Firmware.

5.3.2.1. Memory Management

Platform Firmware manages available memory resources in the most optimal way possible. Since Platform Firmware is aware of memory types (such as RO or RW) and their latencies, the firmware abstracts this complexity from Application Firmware. As a result, Application Firmware can execute on HyperRAM, TCM, or XIP without any functional differences.

5.3.2.2. Host communication

Platform Firmware supports multiple host communication interfaces but provides a single consistent API for Application Firmware to communicate with its counterpart running on the host. This abstraction eliminates the need for Application Firmware to manage active interfaces between the host and GARD hardware.

5.3.2.3. Camera Management

Platform Firmware supports configuring cameras from different vendors and models by accepting camera-specific configurations. These configurations contain information required to set up the camera sensor for parameters such as exposure, resolution, white balance, and other settings.

5.3.2.4. ISP Management

Platform Firmware configures and controls the ISP engine embedded within GARD. The firmware triggers the ISP engine to capture images based on the current pipeline state.

5.3.2.5. Rescale Engine Management

Platform Firmware configures and controls the Rescale Engine embedded within GARD. This engine is triggered by Platform Firmware depending on the pipeline state.

5.3.2.6. ML Management

Platform Firmware allocates space for input and output buffers required by the executing ML network. It uses the remaining available memory (accessible to the ML engine) to hold ML networks during execution. Platform Firmware also starts the ML engine according to the pipeline stage. Once the ML engine completes execution, post-processing logic in Application Firmware is executed.

5.3.2.7. RFS Support

Platform Firmware provides a rudimentary file system, named RFS, for storing firmware, camera configurations, and ML networks on SPI Flash. RFS currently supports routines to read and write modules on SPI Flash.

5.3.2.8. Pipeline Support

Platform Firmware supports a mechanism to define the pipeline that dictates the sequence of executing various engines. This sequence can be modified dynamically based on Application Firmware requirements.

5.3.2.9. Upgrade Support

Platform Firmware supports upgrading the running firmware with another (or the same) version. Although basic, this functionality allows a different version to run on the same hardware.

5.3.3. Operation

The following figure shows the operational flow of the Platform Firmware.

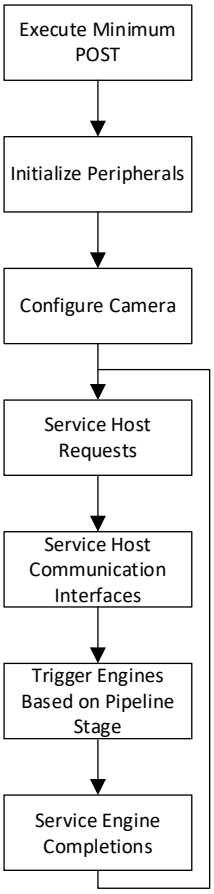

Figure 5.3. Platform Firmware Flow

The Platform Firmware starts by executing a minimal POST to validate the hardware before a complete run. Once the minimal POST is complete, the necessary hardware peripherals, such as OSPI, UART, I2C, GPIO, and ISP, are initialized. This ensures that firmware modules dependent on these peripherals can run without issues.

After the peripherals are ready, the camera configuration is read from RFS and sent to the camera for initialization. At this point, initialization is complete, and the Platform firmware enters its service loop. This loop runs for as long as GARD is powered.

The functions dispatched in this loop perform the following tasks:

- Service host commands
- Manage the pipeline
- Manage ISP
- Manage ML engine
- Run post-processing module

5.3.3.1. Service Host Commands

The HUB running on the host can request the following services from GARD:

- Discovery – The HUB sends the Discover command during its initialization. GARD responds with the string *I AM GARD* to indicate its presence.
- Write/read register in GARD – The HUB sends these requests to write or read registers in the AXI space of GARD.
- Send/receive data from GARD – The HUB sends these requests to write or read memory space in GARD.
- Send application data to HUB – When Application Firmware needs to send data to its counterpart running on the HUB, it calls a function provided by Platform Firmware. This function sends the data to the HUB, which then forwards it to the application waiting on the HUB for this data.

5.3.3.2. Manage Pipeline

The current Platform Firmware implementation supports a static pipeline.

5.3.3.3. Manage ISP

Platform Firmware provides the following ISP management functions:

- Start or stop the ISP engine.
- Configure ISP parameters to crop and/or scale the input image.
- Configure ISP to output the image to a specific memory address.

5.3.3.4. Manage ML Engine

Platform Firmware provides the following ML management functions:

- Start the ML engine.
- Set up the address of the ML network that executes on the ML engine.

5.3.3.5. Run Post-Processing Module

Platform Firmware calls the post-processing routine provided by Application Firmware. This routine is tuned to work with the ML network executed on the ML engine and is aware of the inference data output by the ML network.

6. Developing and Deploying AI Model on FPGA

This section provides an overview of the workflow for developing and deploying AI models on the CertusPro-NX platform. It covers key steps involved in preparing, training, and compiling models for FPGA-based acceleration. The focus is on vision applications such as Multi-Object Detection (MOD) and Defect Detection (DD), which require efficient execution on hardware while maintaining flexibility and performance.

Running AI models on FPGA combines hardware-level acceleration with security and adaptability. The following sections describe the process of creating and deploying models using the sensAI solution stack, which includes tools for training, quantization, simulation, and compilation. For additional details on model architectures and training processes, refer to the Lattice sensAI Studio documentation.

6.1. sensAI Solution Stack

The sensAI solution stack provides a suite of tools designed to facilitate the training of neural network models optimized for FPGAs. These tools are organized into three Python packages:

- sensAI Neural Network Training Environment (LATTE) – A framework designed to train deep neural networks. For more information, refer to the Lattice sensAI Studio documentation.
- sensAI Neural Network Quantizer (LSCQuant) – Provides tools for quantizing neural network models for FPGA deployment. It integrates with LATTE to enable Quantization-Aware Training (QAT). For more information, refer to the Lattice sensAI Studio documentation.
- sensAI ML Engine Simulator – Allows trained neural network models to execute in an FPGA-like environment. This enables LATTE to perform post-training tests and evaluate model performance without requiring physical FPGA hardware.

6.2. Preparing AI Model

FPGAs impose constraints on model architecture, including limitations on supported layers and specific layer combinations. Input size is also restricted and may vary depending on whether external memory is used. Detailed information on these architectural, layer, and input size limitations is available in the sensAI SDK documentation.

The model must be defined using the Keras Functional API. Typically, this is implemented as a Python function that accepts hyperparameters as arguments and returns a Keras functional model. The LSCQuant package provides pre-built layer blocks that can be used to construct models based on well-known architectures such as MobileNet.

The number of samples required to train a model depends on several factors, including task complexity and whether the model is trained from scratch or fine-tuned:

- A model trained from scratch typically requires at least 100,000 labeled samples.
- Fine-tuning a model generally requires at least 10,000 labeled samples.

A portion of these samples should be reserved for a test dataset to evaluate the model after training or fine-tuning, usually between 10–20% of the total. It is also common practice to set aside some samples from the training dataset for validation during training.

Python code must be written to:

- Load the dataset.
- Provide appropriate inputs to the model during training.
- Format associated labels.
- Optionally perform data augmentation (for example, random changes in brightness or contrast, rotations, and translations).

When using a training framework such as LATTE, data loading is typically implemented as a Python class that inherits from one of the dataset handler classes provided by the framework.

6.3. Training AI Model

Training or fine-tuning a model for FPGA deployment requires QAT techniques. Using LATTE together with LSCQuant significantly simplifies this process.

LATTE requires a combination of Python code and a configuration file to train a model. The Python code typically includes:

- Model function – A function that accepts model hyperparameters as input and returns a Keras Functional model.
- Dataset handler class – Responsible for loading image samples, formatting labels, and performing data augmentation. This handler can be shared across training, evaluation, and testing datasets, or separate handlers can be defined for each.
- Python module – Defines (or imports) the model function and dataset handler class(es) and registers them with LATTE.

For each training experiment, LATTE requires a configuration file in YAML or JSON format. This file specifies:

- Registered model and dataset handlers, along with their hyperparameters (for example, dataset path and augmentation settings).
- Key training parameters such as the number of epochs, learning rate, optimizer, and Keras losses and metrics (or any custom metrics registered).
- Whether the model should be trained with QAT using LSCQuant, along with the chosen quantization scheme.
- If fine-tuning a pre-trained model, the path to the pre-trained weights.

6.3.1. Training Command

To start the training, use the following command:

```
latte train experiment-1.yaml code.py
```

Where:

- *experiment-1.yaml* is the configuration file.
- *code.py* is the Python module that registers the model function, dataset handlers, and any custom losses or metrics into LATTE.

6.3.2. Training Process

The training process can take anywhere from a few minutes to several hours or even days, depending on model complexity and the dataset size. During training, LATTE creates an artifact folder named after the configuration file (excluding the extension) to store all training outputs, including checkpoints and logs.

If training is interrupted, either manually (for example, using CTRL + C) or due to a process termination, it can be resumed from the last saved checkpoint by simply rerunning the same command.

6.3.3. Evaluating AI Model

Once training is complete, evaluate the trained model on the test dataset using the following command:

```
latte test experiment-1.yaml code.py
```

The two files used for testing are the same as those used for training. By default, LATTE selects the best checkpoint, the one corresponding to the epoch with the lowest loss or highest evaluation metrics during training. The configuration file can also specify testing using the ML engine simulator, which executes the model as if running on an FPGA.

6.3.4. Converting AI Model

Convert the trained model into a file format compatible with the sensAI SDK compiler by using the following command:

```
latte convert experiment-1.yaml code.py -c sensai-h5
```

Similar to testing, the conversion process uses the model weights from the best checkpoint by default. The output is an H5 file stored in the convert folder within the artifact directory (for example, experiment-1/convert). You can also specify the conversion format in the configuration file to avoid including the *-c sensai-h5* argument in the command line.

To access the LATTE tool and learn more about training the MOD and DD models used in the demo, contact senseaiaccess@latticesemi.com.

6.4. Compiling AI Model for FPGA

Lattice sensAI Studio is used to compile models for deployment on the CertusPro-NX platform. Compilation can be performed using either the GUI or command-line interface to generate the output model.

6.4.1. Using GUI

1. Open the Lattice sensAI software.
2. Start a new project by navigating to **File** > **New**.
3. Configure the project as follows:
   - Enter a project name.
   - **Framework**: *Keras*
   - **Class**: *CNN*
   - **Device**: *Certus-Pro NX*
   - **IP**: *Advanced_CNN*
   - Click **Network File** and select the *.h5* network file.
   - Click **Image/Video/Audio Data** and select an image (required for the process, but output is unused).
4. Click **NEXT**.

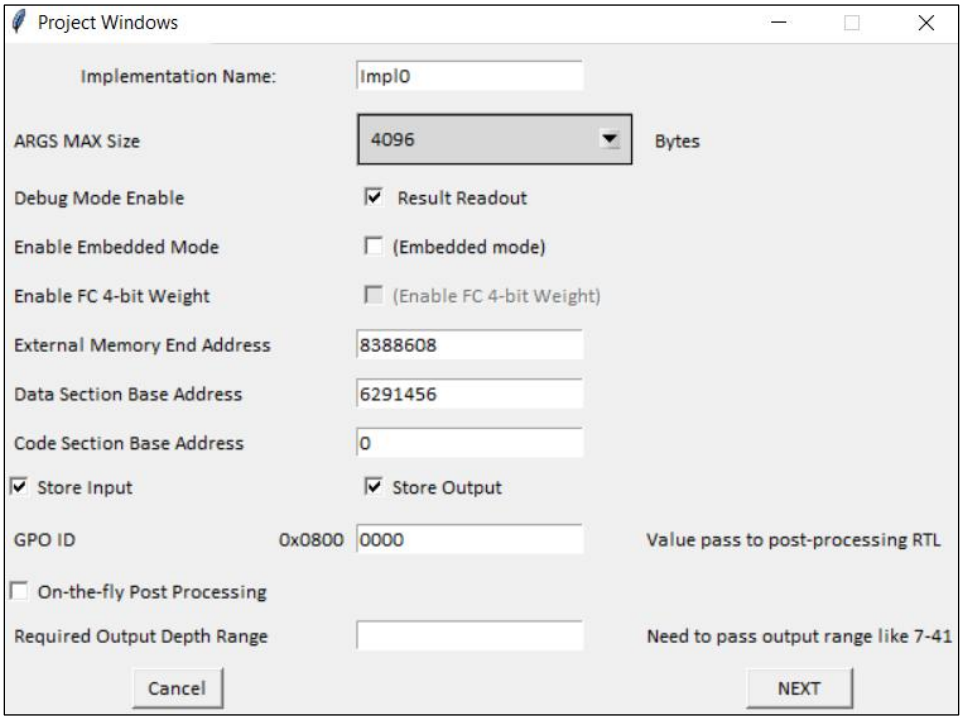

Figure 6.1. Implementation Configuration Window

5. In the Implementation window, check the **Store Input** and **Store Output**. Leave all other values as populated, then click **NEXT**.

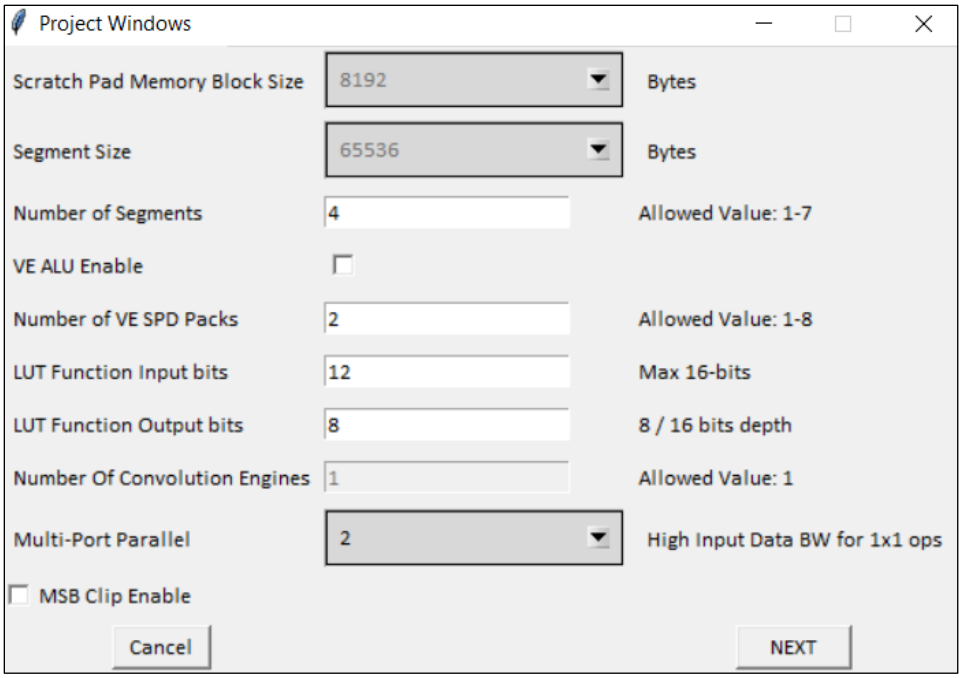

Figure 6.2. Second Implementation Options Window

6. In the second implementation window, leave all values as they are (populated by the network). Click **NEXT**.

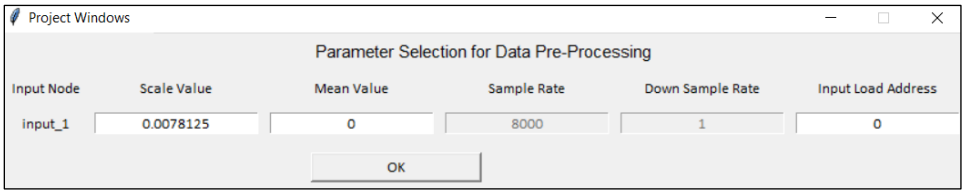

Figure 6.3. Parameter Selection Window

7. In the Parameter Selection window, leave all values as default. Click **OK**.
8. Click **Process** on the toolbar and select **Analyze**. This process takes approximately 5 minutes.

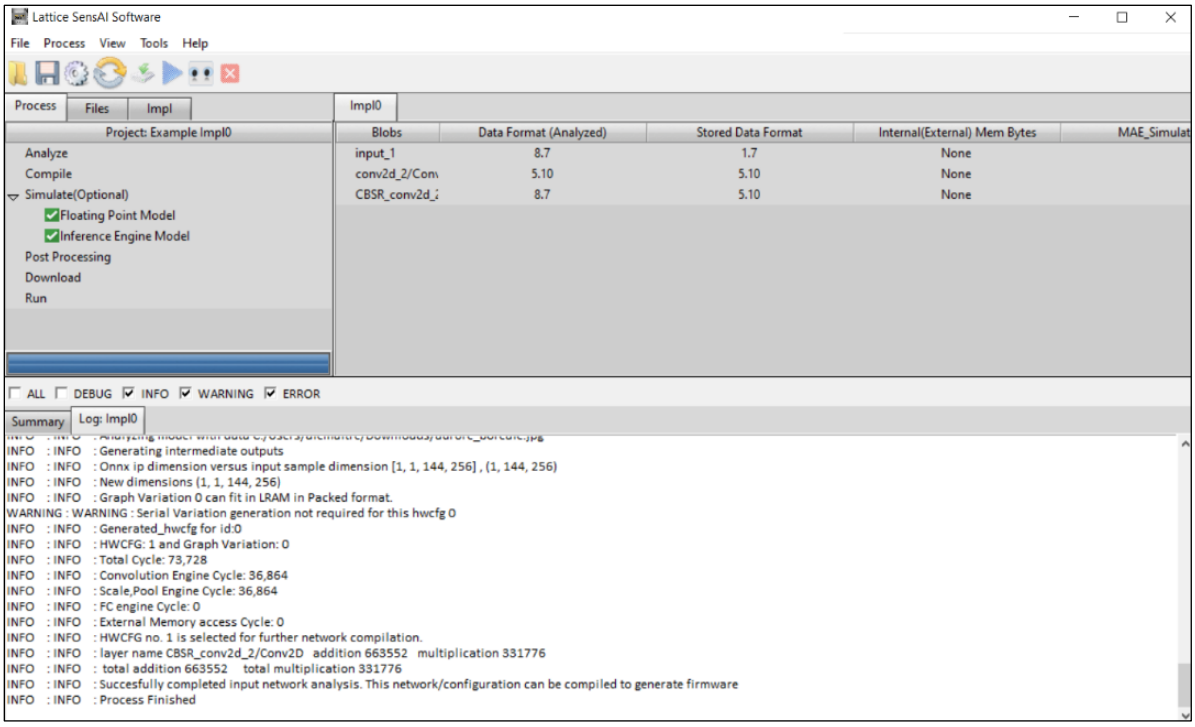

Figure 6.4. Analyze Step Complete

9. After analysis, click **Process** on the toolbar and select **Compile**. This process takes approximately 5 minutes.

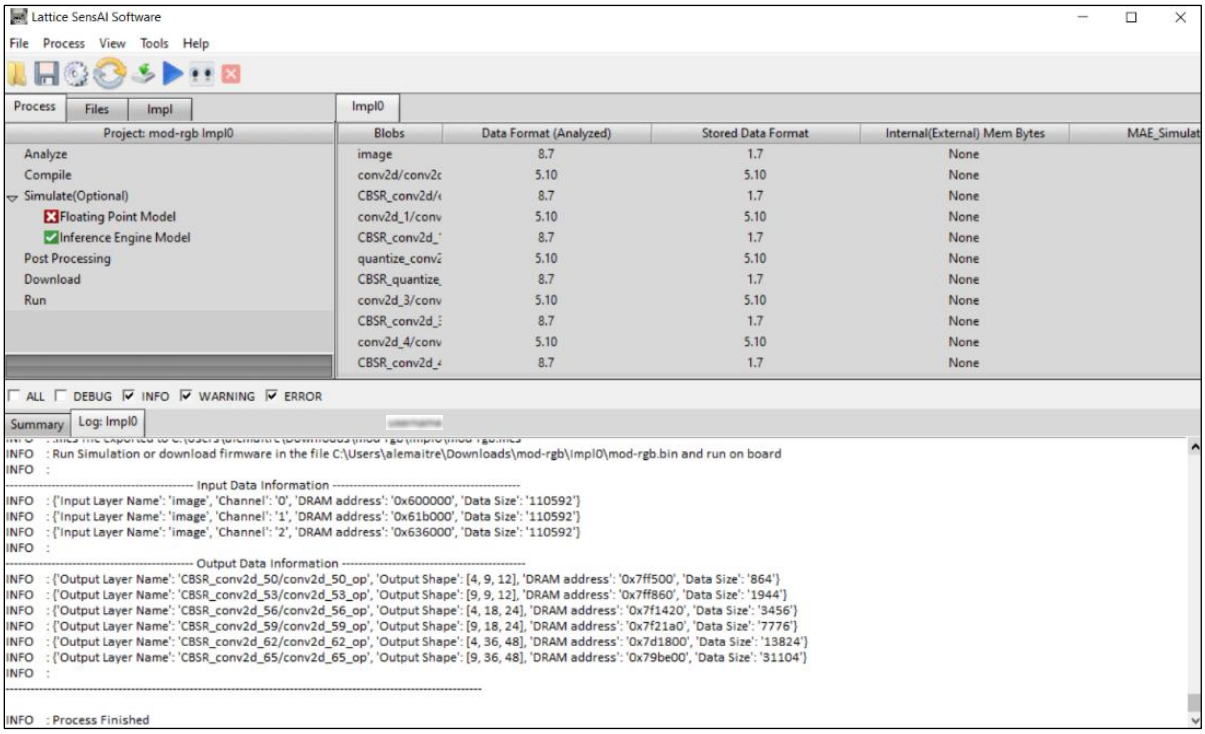

Figure 6.5. Compile Step Complete

10. Note the **DRAM address** and **Data Size** from the output. These values may be used to identify memory regions that should remain reserved. Locate the output folder and download the *.bin* file. Combine this file with the FWRoot file using the *RootimageBuilder.py* script provided.

6.4.2. Using Command Line

You can use the following **Analyze** and **Compile** scripts instead of the GUI:
Note: Adapt the variables to match your files and their paths.

- Analyze Script:

```
C:\lscc\ml\8.0\win64\lsc_ml_compl.exe --cmd analyze --project_name test --project_dir
c:\users\*****\Downloads\ --network_file "c:\users\*****\Downloads\mod-cpnx-8.2.0 1.h5" --
image_files "c:\users\*****\Downloads\toronto_street_384x288x3.png" --device "Certus-Pro NX" --
ip_mode "Advanced_CNN" --framework Keras --num_conv_eng 1 --num_ebr 4  --ebr_blk_size 2048 --
arg_max_size 4096 --lut_input_bits 12 --lut_output_bits 8 --msb_clip_enable 0 --ve_alu_enable 0 --
num_ve_spd_packs 2 --num_ve_segments 4 --multi_port 2 --segment_size 65536 --enable_debug_mode 1 -
-extmem_size 8388608 --crosslink_code_base_addr 0 --crosslink_data_base_addr 6291456 --
crosslink_lram_size 262144 --crosslink_scratch_pad_blk_size 8192 --num_ebr 4 --input_ebr "" --
output_ebr "" --mean 0 --scale 0.0078125 --extmem_off 0 --enable_dualcore 0 --enable_quadcore 0 --
enable_embedded_mode 0 --enable_fc_4_bit_weight 0 --reg_out 0 --load_from_extmem 1 --
store_to_extmem 1 --required_output_depth_range ""
```

- Compile Script:

```
C:\lscc\ml\8.0\win64\lsc_ml_compl.exe --cmd compile --project_name test --project_dir
c:\users\******\Downloads\ --network_file "c:\users\*****\Downloads\mod-cpnx-8.2.0 1.h5" --
image_files "c:\users\*****\Downloads\toronto_street_384x288x3.png" --device "Certus-Pro NX" --
ip_mode "Advanced_CNN" --framework Keras --num_conv_eng 1 --num_ebr 4  --ebr_blk_size 2048 --
arg_max_size 4096 --lut_input_bits 12 --lut_output_bits 8 --msb_clip_enable 0 --ve_alu_enable 0 --
num_ve_spd_packs 2 --num_ve_segments 4 --multi_port 2 --segment_size 65536 --enable_debug_mode 1 -
-extmem_size 8388608 --crosslink_code_base_addr 0 --crosslink_data_base_addr 6291456 --
crosslink_lram_size 262144 --crosslink_scratch_pad_blk_size 8192 --num_ebr 4 --input_ebr "" --
output_ebr "" --mean 0 --scale 0.0078125 --extmem_off 0 --enable_dualcore 0 --enable_quadcore 0 --
enable_embedded_mode 0 --enable_fc_4_bit_weight 0 --reg_out 0 --load_from_extmem 1 --
store_to_extmem 1 --required_output_depth_range ""
```

6.5. Programming and Testing AI Model on FPGA

For operating the HUB and viewing the model output, refer to the following demo guides:

- CertusPro-NX Multi-Object Detection on SOM Board Demo Guide (FPGA-UG-02247)
- CertusPro-NX Defect Detection on SOM Board Demo Guide (FPGA-UG-02248)

7. HUB and Host Application

Host-accelerated Unified Bridge (HUB) is a companion software stack that provides a feature-rich library to interface with GARD firmware, making effective use of multiple channels for efficient control and data exchange. HUB is designed with a robust, enterprise-grade architecture that emphasizes interoperability, scalability, and high availability for peak performance and operational continuity.

HUB is built to address the requirements of multiple domains such as industrial, automotive, aerospace and defense, and server/rack management solutions.

As a userspace-facing shared library, HUB uses a mix of HUB OS Linux device drivers to connect to GARD:

- Inbox (standard RPi Linux): I2C, UART, GPIO, and MIPI CSI-2.
- Customized: FPGA-attached (MIPI) cameras, PCIe endpoints, USB interfaces, and on-board sensors.
- Enhanced: Kernel and userspace frameworks for collecting AI/ML metadata interleaved in the MIPI CSI-2 bitstream.

HUB functions as an intelligent multi-protocol gateway and hardware abstraction layer, providing a stable, unified interface between ARM-Linux and the real-time GARD firmware, and abstracting the host application from hardware complexities.

HUB leverages the Rich-OS (ARM Linux) ecosystem, enabling integration with industry-standard software stacks:

- V4L2 (Video4Linux2) – A framework that handles multimedia devices in Linux, including cameras, sensors, and displays.
- libcamera – An open-source software for image signal processing on embedded cameras.
- OpenCV – A cross-platform software, GPU-accelerated (where applicable) library of programming functions aimed at computer vision and image/video processing.
- GStreamer – A pipeline-based multimedia framework that links together various media processing systems to create complex multimedia workflows.
- Python libraries and frameworks – Embedded web servers (Flask), graphing and visualization (Plotly and Matplotlib), and data processing libraries (NumPy).

HUB coordinates with GARD firmware to stream GARD application metadata (such as bounding-box coordinates and class confidence scores) to the host application, through a callback registration mechanism. This enables real-time visualization of AI/ML metadata integrated with live sensor feeds.

HUB packages C and Python libraries with demonstrative host applications. HUB ships with PyHUB – a HUB Python library with C bindings for the core HUB library. PyHUB enables Python application development by abstracting low-level hardware and driver complexity, allowing the GARD platform to leverage the underlying FPGA pipelines using high-level Python code. This approach optimizes the AI model development, testing, and deployment cycle.

Additional information about HUB is available on the sensAI Reference Design GitHub page.

7.1. HUB Release Artifacts

The HUB OS is a customized 64-bit ARM Linux operating system image (currently based on Raspbian Bookworm 12).

The lscc-hub packages deliver the core HUB libraries and tools for application development on HUB OS:

- lscc-hub Debian Package: *libhub.so* and *hub.h* – ARM64 Linux shared library and header file.
- lscc-hub Wheel Package: PyHUB (*hub.py*) – Python library with C-bindings to *libhub.so* for creating Python Applications.

While HUB is delivered as a shared library (.so), the released codebase also includes sample applications demonstrating API usage (*hub_apps*):

- Minimal Application (*app.c*) – Creates a *minimal* application that demonstrates HUB initialization, GARD handling, GARD operations such as register reads/writes and bulk reads/writes, sensor data reading, HUB cleanup, and shutdown/exit.
- Streaming Application (*streaming_app.c*): Creates a *streaming* application that demonstrates HUB initialization, GARD handling, and streaming of application-specific data (*AppData*) originating from *GARD_FW* (in the GARD ML engine) to the host application running on HUB through user-supplied callback functions.

- Image Operations Application (*img_ops_app.c*): Sends a command to GARD Firmware to capture a rescaled image from the ISP, read it from GARD HyperRAM, and to save it locally as a BMP/JPG/PNG/RAW file.
- EdgeHUB: a Python Visualizer web application, which is detailed in the Host Application section.

HUB provides versioned installer packages that follow the *major.minor.bugfix* pattern for installation on HUB OS:

- *lscc-hub_<version>_arm64.deb* – Debian Package.
- *lscc_hub-<version>-py3-none-manylinux_2_28_aarch64.whl* – Wheel Package.

7.2. HUB Platform Support

HUB supports *TORNA CPNX SOM MODULE B02 REVB-05092025*.

The following table lists hardware interfaces supported by HUB (through HUB OS).

Table 7.1. HUB Interfaces

Interface	Speed/Frequency	Connected Endpoints	Usage
I2C	100 kHz	Raspberry Pi CM5 and GARD	GARD discovery, control, and data transfers.
UART	921600 baud	Raspberry Pi CM5 and GARD	GARD discovery, control, and data transfers.
UART	115200 baud	Raspberry Pi CM5 and debug port of user machine	Raspberry Pi OS Linux login shell.
MIPI CSI-2	2 lanes at 912 Mbps/lane 4 lanes at 720 Mbps/lane	Raspberry Pi CM5 and GARD	High-speed transfer of camera raw frame feed.
GPIO	—	Raspberry Pi CM5 and GARD	Asynchronous interaction with GARD.
Ethernet	1 Gbps	Wired network-capable device	High-speed network data transfers.
USB	3.0	Raspberry Pi CM5 and USB devices	Storage device file-system access.
SPI (planned)	49MHz (per lane)	Raspberry Pi CM5 and GARD	Medium-speed data transfers.
PCIe (planned)	Gen2 ×1 (5 GT/s)	Raspberry Pi CM5 and GARD	High-speed data transfers.

7.3. HUB Features

The following table lists features supported by HUB.

Table 7.2. HUB Features

Feature	Support	Description
Host configuration JSON parsing	Parsing host config (bus info)	Pre-init
GARD discovery	Supported buses: I2C and UART	Discovery handshake with connected GARD
GARD profile parsing	GARD profile ID	Canned GARD profile ID
GARD data movement (32-bit register read-write)	Supported buses: I2C and UART	—
GARD data movement (buffers/chunks)	Supported buses: I2C and UART	—
On-board sensor data acquisition	Supported platform: SOM and carrier board	TMP118 (temperature), INA236 (energy)
IMX219 camera support	Live video feed from connected IMX219 camera in 2-lane and 4-lane configurations	—
Application metadata streaming	Metadata streaming application from GARD to host application using callbacks	Sample C and Python callback functions provided
Rescaled image capture	Capture of rescaled image from GARD ISP	—
Python library	HUB and GARD classes: • Read-write operations on registers • Read-write operations on bulk data	—

Feature	Support	Description
	• Sensor data collection API • Application metadata streaming API	
EdgeHUB – web application	Multi-tab web application displaying: • Real-time visualization and trending of on-board sensor data • Live video feed from attached IMX219 (MIPI) camera • Image Ops pipeline • Live video feed from CrossLink-NX SOM camera (UVC)	—
Packaging	Versioned ARM64 Debian Package Versioned Python wheel Package	Installs in/opt/hub

7.4. HUB Operational Flow

HUB primarily interacts with the GARD firmware running on the GARD instantiated in the CertusPro-NX FPGA.

The following sequence diagrams represent various interactions between HUB and GARD Firmware:

- HUB initialization:

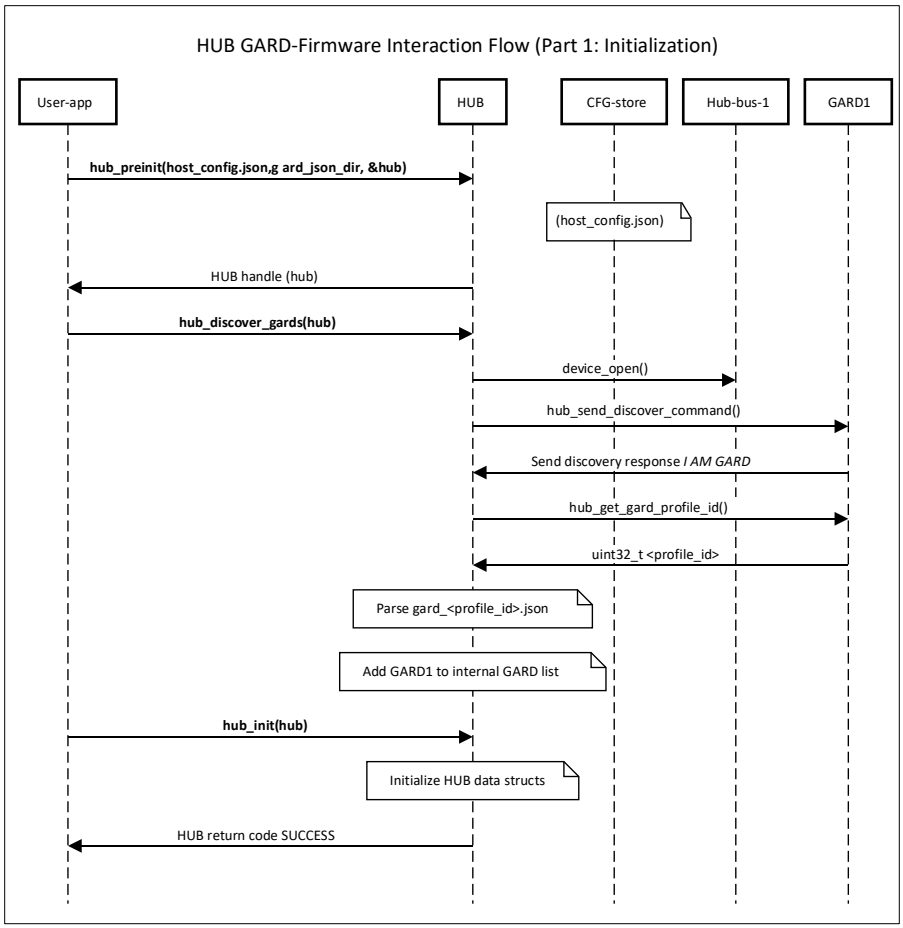

Figure 7.1. HUB GARD-Firmware Initialization

- GARD selection and operations:

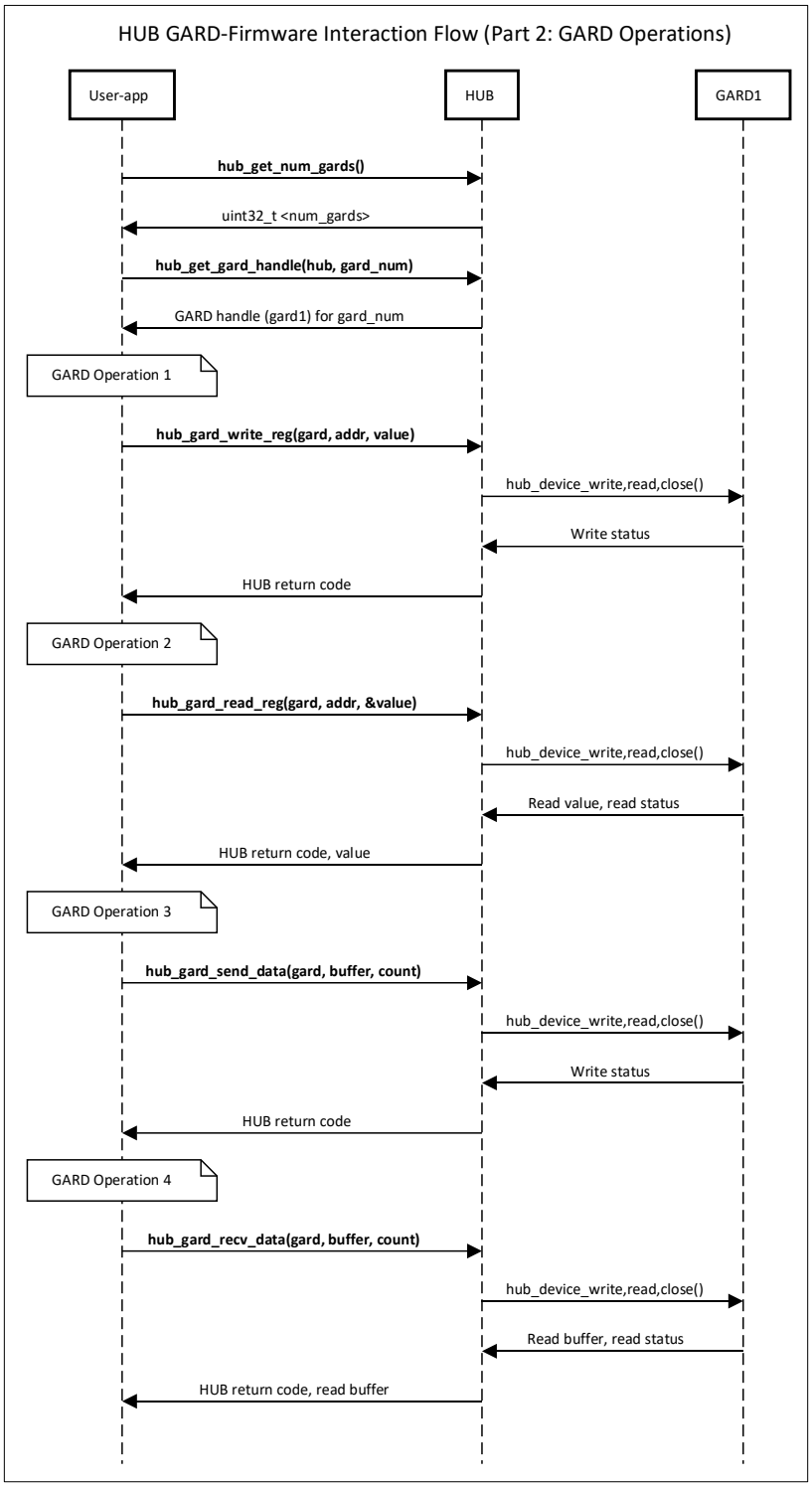

Figure 7.2. HUB GARD-Firmware Operations

- On-board sensor data acquisition:

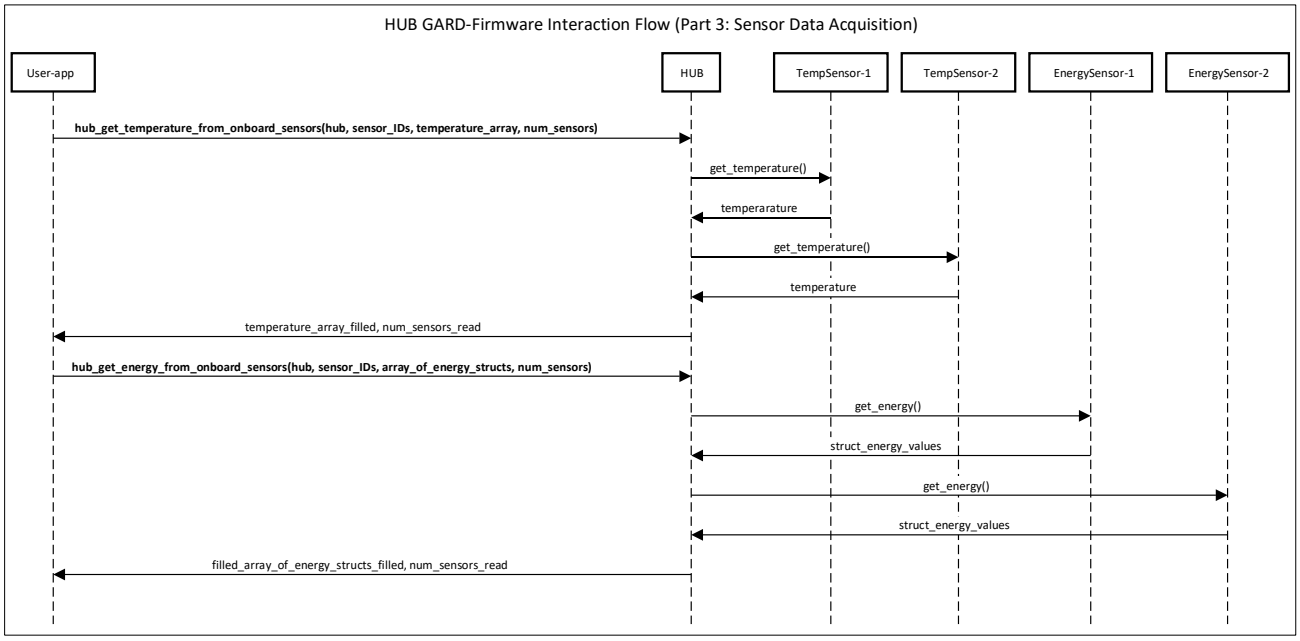

Figure 7.3. HUB GARD-Firmware Sensor Data Acquisition

- HUB application data streaming (currently uses one GPIO pin and I2C bus):

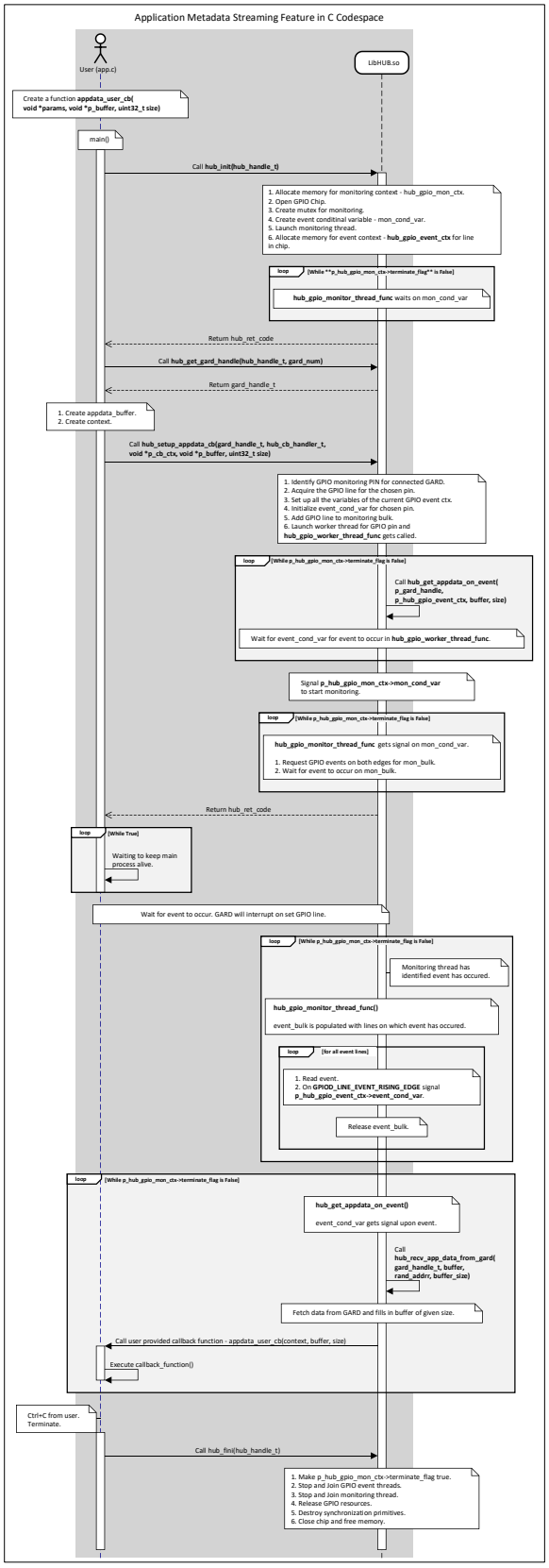

Figure 7.4. HUB Application Data Streaming Feature

- HUB shutdown and cleanup:

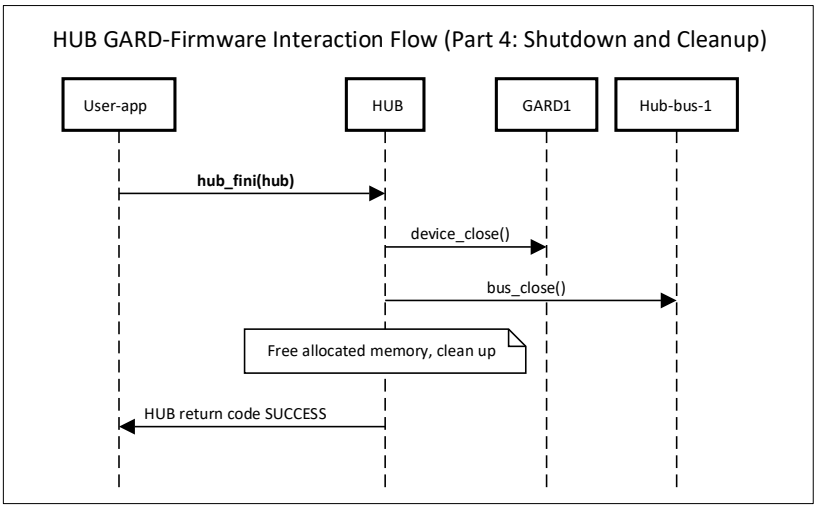

Figure 7.5. HUB GARD-Firmware Shutdown and Cleanup

7.5. Host Application

HUB ships with EdgeHUB, an inbuilt diagnostic visualizer and a multi-tabbed web application featuring:

- Real-time Sensor Data Graphing
  The following figure shows an exemplary browser screenshot of the sensor data acquisition feature in action. Four graphs are displayed, each focusing on a specific quantity: temperature, voltage, current, and power. Corresponding numerical values appear in the sidebar. The update frequency is 1/s.

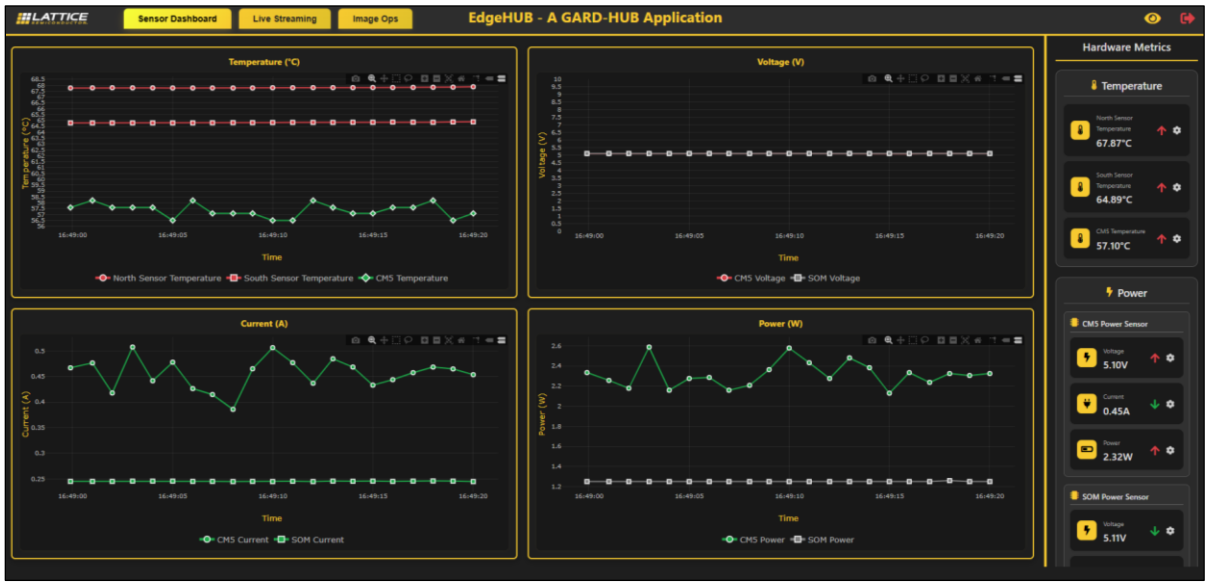

Figure 7.6. Sensor Data Graphing

- Live Video Stream (from GARD-connected MIPI camera and/or CrossLink-NX UVC camera)
  The following figure shows a tab with a live video feed from a connected MIPI IMX219 camera. EdgeHUB can also display live feed from a CrossLink-NX UVC camera.

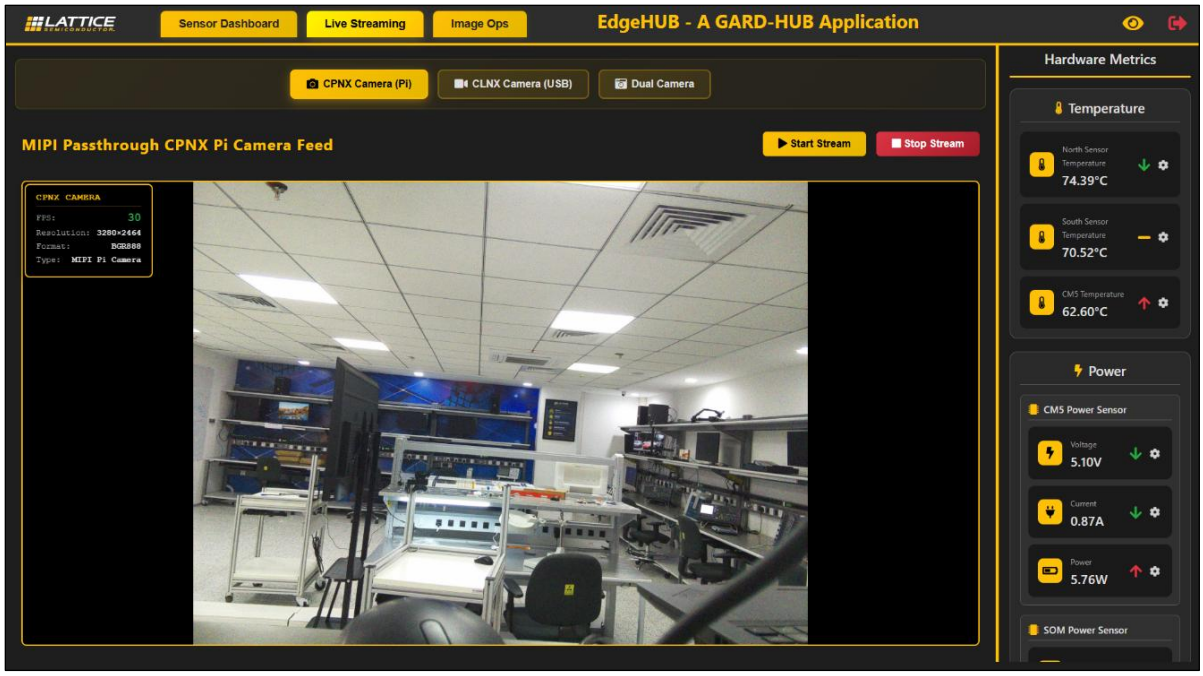

Figure 7.7. Live Camera Feeds

- GARD-ISP Image Capture and Processing (E2E Image Ops Pipeline)
  The following figure shows the **Image Ops** tab. EdgeHUB enables capture and display of an image from the GARD ISP. You can save the displayed image either locally (on Raspberry Pi CM5) or on your machine (PC/laptop) in PNG, BMP, JPG, and RAW formats.

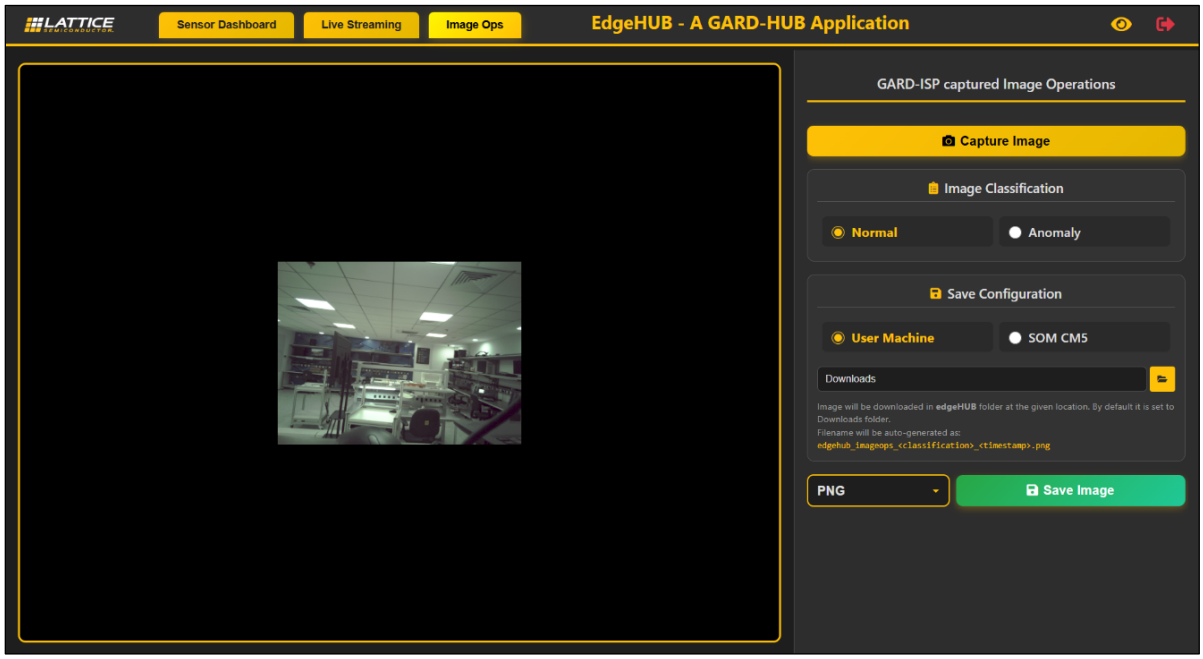

Figure 7.8. Image Ops Pipeline

The synchronous views from the **Live Streaming** and **Image Ops** tabs enable developers to correlate live camera input with AI model output, turning complex integration challenges into rapid, visual debug cycles.

For instance, when an AI model misclassifies an object, you can instantly correlate the model output with the visual feed and upstream image processing parameters (ISP-controlled, such as color or lighting issues), leading to accelerated root-cause isolation, analysis, and resolution.

EdgeHUB functions as a robust framework for advanced debugging and hardware-in-the-loop (HWiL) simulation.

The integrated web server provides networked access, enabling remote management and debugging of FPGA hardware and transforming the GARD assembly into a comprehensive validation environment.

The rigid control flow established by the HUB architecture decouples the host application's software lifecycle from FPGA hardware changes. This architecture also establishes a secure gateway designed to support Secure Over-The-Air (S-OTA) updates, ensuring AI models and firmware patches are delivered securely and reliably throughout the device's operational lifecycle.

8. Programming and Running Demos

For programming and running the demo procedures, refer to the following demo guides:

- CertusPro-NX Multi-Object Detection on SOM Board Demo Guide (FPGA-UG-02247)
- CertusPro-NX Defect Detection on SOM Board Demo Guide (FPGA-UG-02248)

Appendix A

A.1. GARD Firmware Setup

A.1.1. Development Environment

To develop GARD platform firmware efficiently, you need a properly configured environment with essential tools and SDKs. This section outlines the recommended operating system, IDE, and supporting utilities required for building, debugging, and flashing firmware.

The list of software used, including minimum versions and associated extensions (if applicable), is as follows:

- Microsoft Windows OS
- Microsoft Visual Studio Code (v1.98.2 or later) with the following extensions:
  - C/C++ Extension Pack (v1.3.1 or later) – This extension adds other extensions that make C/C++ code development much easier.
  - Makefile Tools (v0.12.17 or later) – This extension helps build and run programs from within the IDE.
  - Hex Editor (v1.11.1 or later) – This extension provides an interface to view and edit file content in hexadecimal form.
  - Python (v2025.18.0 or later) – This extension helps write and debug Python code and is useful for creating or debugging Python scripts.
  - Remote SSH (v0.120.0 or later) – This extension is useful for development when the code is located on a remote machine that can be accessed using SSH.
- Git
- Propel SDK (v2025.1 or later)
- Radiant Programmer (v2025.1.1.249.0 or later)

Notes:

- Linux (for example, Ubuntu) is an alternative environment, but this guide focuses on Microsoft Windows.
- Microsoft Visual Studio Code is recommended but not mandatory.
- Microsoft Visual Studio Code extensions can be installed from the extension tab in the activity bar. To install extensions without internet access, obtain the VSIX packaged extension files and install them using the *Install from VSIX* option in the extension action menu.
- Propel SDK is required for the GNU toolchain (compiler, linker, and debugger).
- Radiant Programmer is required for flashing the bitstream and firmware root image to SPI NOR flash.

A.1.1.1. Supplementary Setup Instructions

This section provides additional setup steps that extend beyond the default setup process.

Microsoft Visual Studio Code

During installation setup, ensure you select the following additional tasks:

- *Register Code as an editor for supported file types*
- *Add to PATH (requires shell restart)*

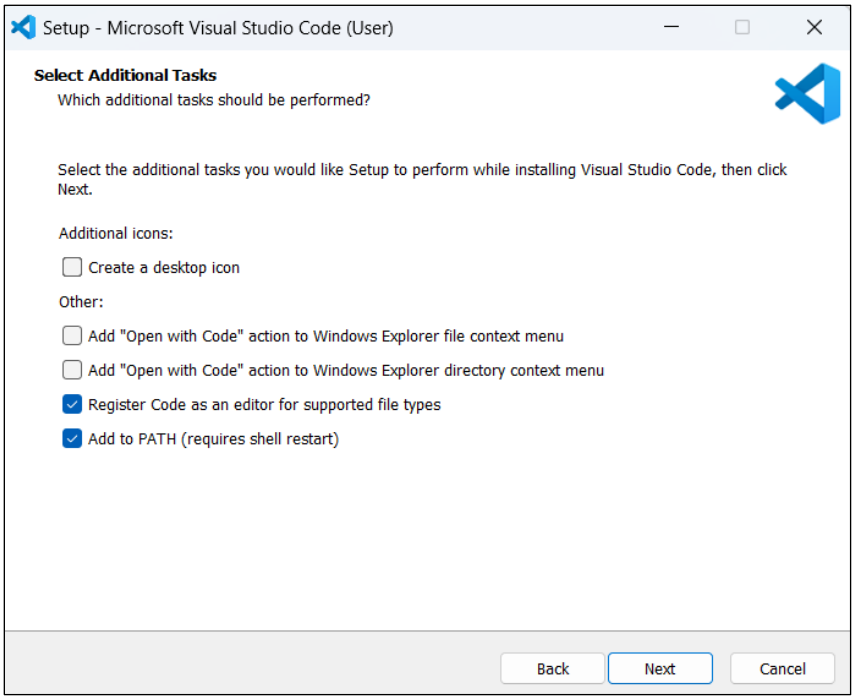

Figure A.1. Select Additional Tasks

Propel SDK

After installation, configure the environment variables by following these steps:

1. Search for Environment Variables in the system settings.
2. Select **Path** and click **Edit**.

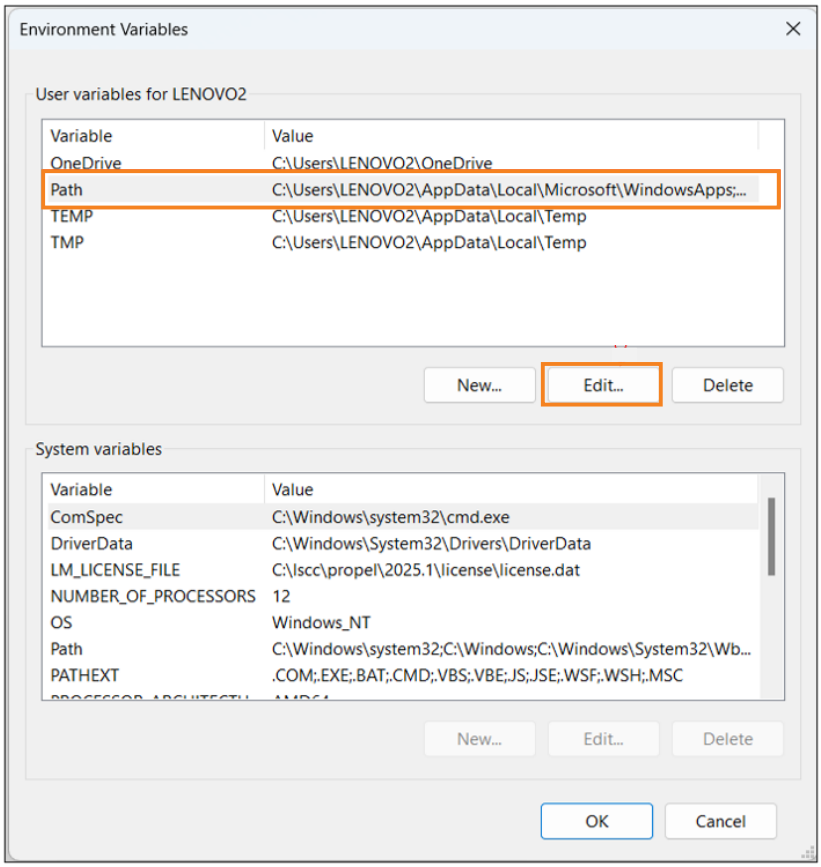

Figure A.2. Environment Variables

3. Add the following paths to the environment variables and click **OK**:
   - *<installation_path>\sdk\build_tools\bin (default: C:\lscc\propel\2025.1\sdk\build_tools\bin)*
   - *<installation_path>\sdk\tools\bin (default: C:\lscc\propel\2025.1\sdk\tools\bin)*
   - *<installation_path>\sdk\riscv-none-embed-gcc\bin (default: C:\lscc\propel\2025.1\sdk\riscv-none-embed-gcc\bin)*

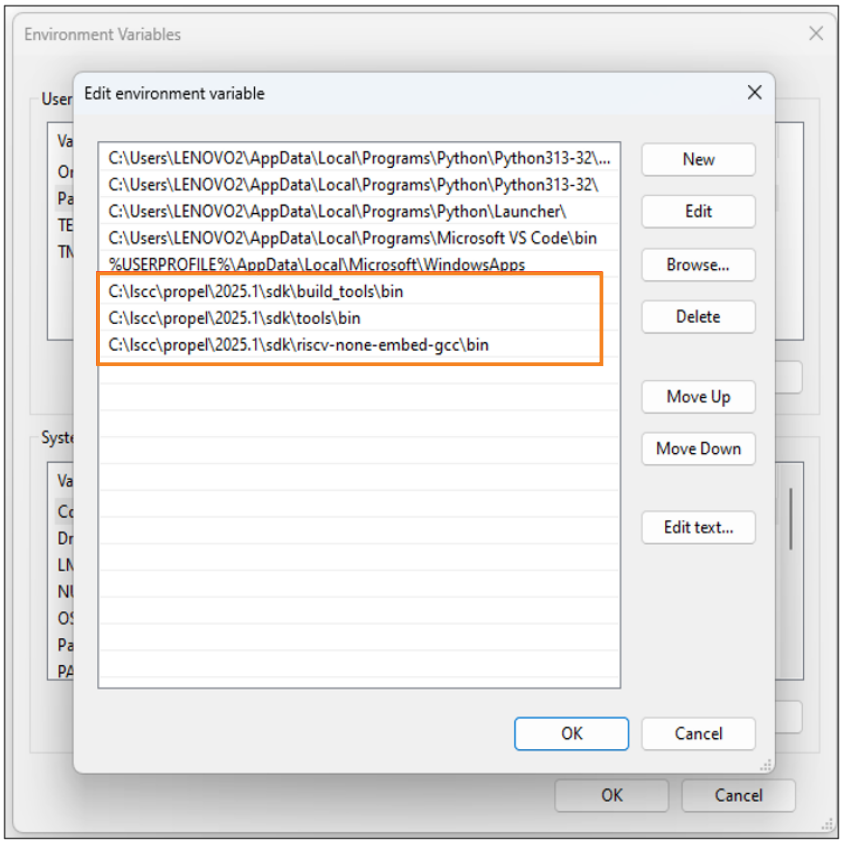

Figure A.3. Edit Environment Variables

Git

During installation setup, ensure you select the following components:

- *Windows Explorer integration*
  - *Open Git Bash here*
  - *Open Git GUI here*
- *Git LFS (Large File Support)*
- *Associate .git* configuration files with the default text editor*
- *Associate .sh files to be run with Bash*
- *Scalar (Git add-on to manage large-scale repositories)*

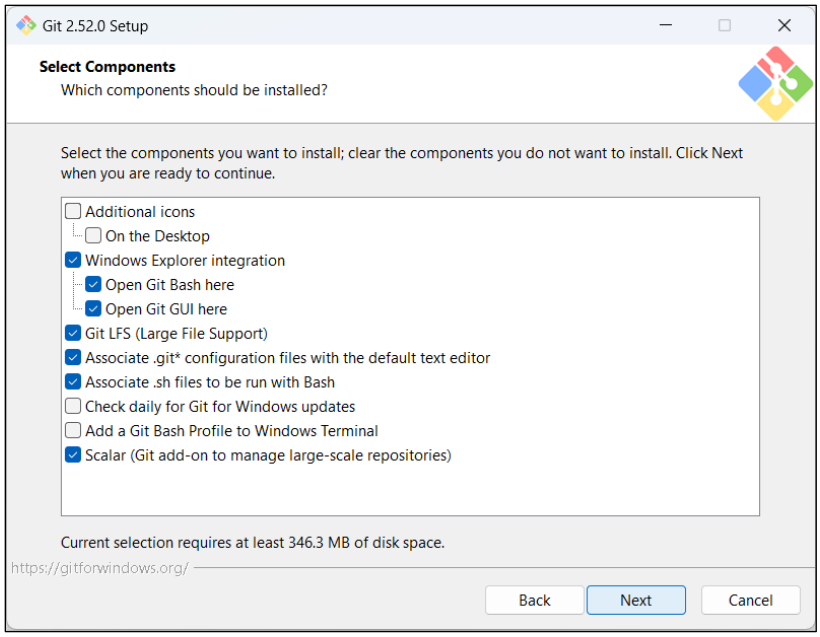

Figure A.4. Select Components

For the **Configuring the line ending conversions** setup, it is recommended to choose **Checkout as-is, commit Unix-style line endings** option.

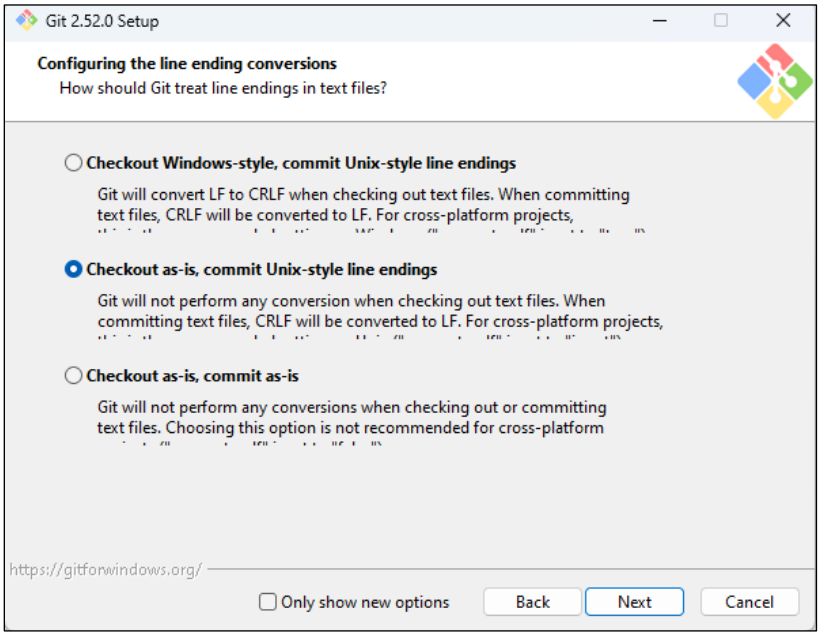

Figure A.5. Unix-Style Line Endings Option

A.1.2. Getting Firmware Source

Before building or modifying the firmware, you must access the official source code. This section explains where to find the latest stable versions of GARD firmware, either from GitHub repositories or SharePoint, and provides guidance on selecting branches, tags, and releases.

A.1.2.1. GitHub

Platform firmware code is available in the GARD Firmware Repository on GitHub.

A.1.2.2. Branches

The following branches host the code in various stages:

- Latest code branch: *fw_core-main*
  The latest GARD platform code resides in the *fw_core-main* branch. The code in this branch is used during development releases for quality assurance. The code in this branch is built up when the feature branches are merged into this branch. Use the code from this branch only when the required feature is not available in one of the release branches.
- Development release branches
  Development release code is stored in branches named using the format: *dev_release/X.Y.Z-GARD-MOD-FW*.
  For example, if the latest development release branch is: *dev_release/2.1.0-GARD-MOD-FW*.
  From this branch name, it can be inferred that:
  - This is a development release.
  - Major release number: 2
  - Minor release number: 1
  - Bugfix release number: 0
  - Contains GARD platform firmware.
  - Compiled for hardware designed to run MOD.

Note: Do no take changes from a development release branch.

A.1.2.3. Release Tags

Officially released code is tagged using the format: *GARD-MOD-FW-vX.Y.Z*.

For example, if the latest tagged code for the GARD firmware is: *GARD-MOD-FW-v2.1.0*.
From this tag name, it can be inferred that:

- The code is tagged and no further commits are accepted under this tag (branch).
- Major release number: 2
- Minor release number: 1
- Bugfix release number: 0
- Contains GARD platform firmware.
- Compiled for hardware designed to run MOD.

Note: It is encouraged to use the tagged release code when needed.

A.1.2.4. Downloading Code from GitHub

Clone the repository to your local machine as follows:

1. Go to the repository Home Page.
2. Click the **Code** dropdown menu.

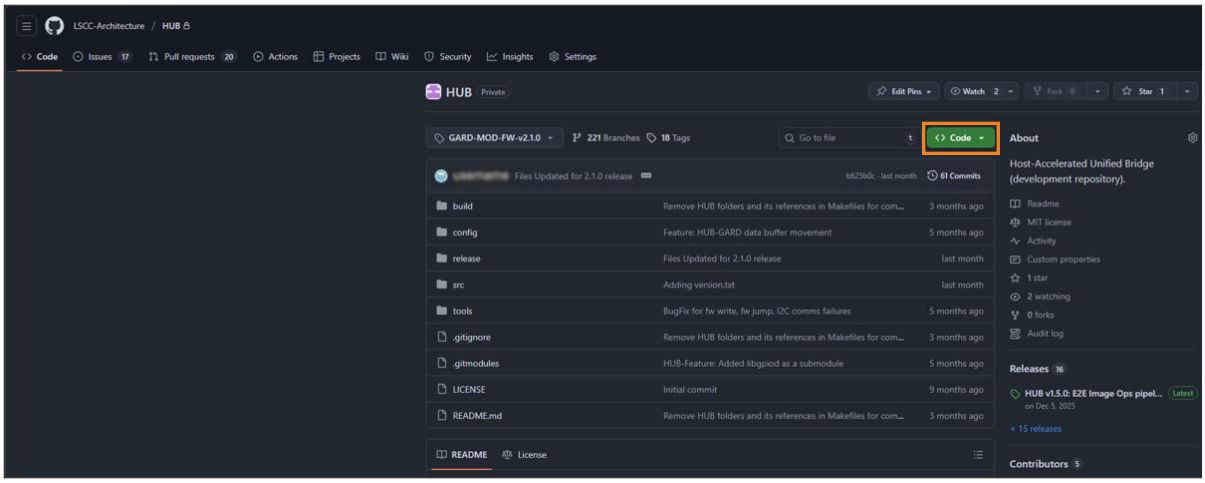

Figure A.6. GitHub Web User Interface

3. Click **SSH** and copy the Git URL. The Git URL is now copied to your clipboard.

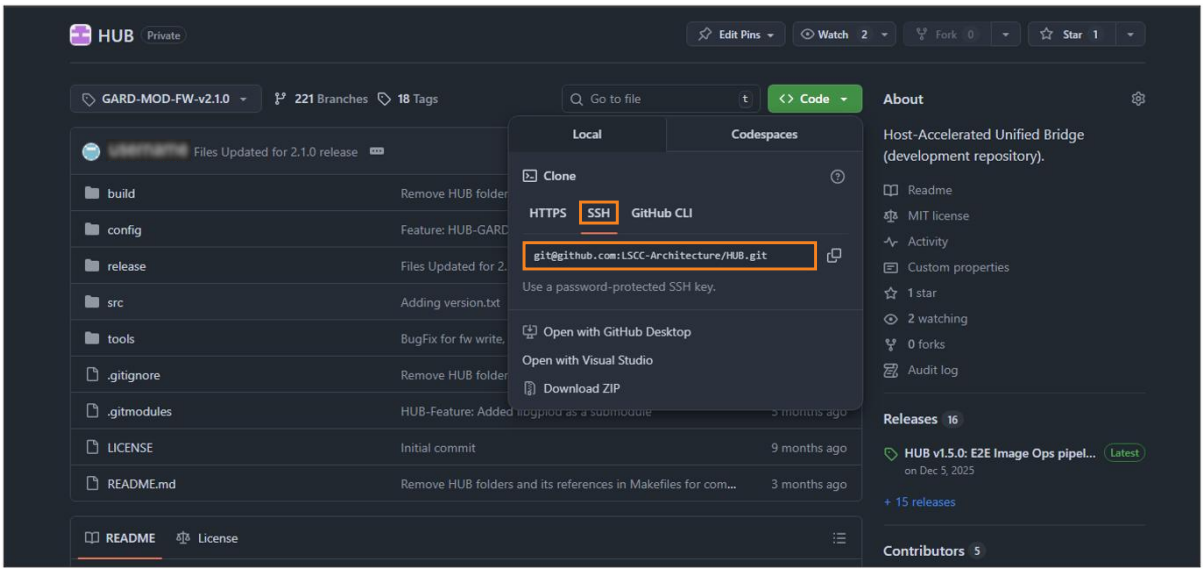

Figure A.7. Copy GitHub URL

4. Launch the **Git Bash** application.
5. Navigate to the folder where you want to clone the Git repository.
6. To clone the entire repository into a folder named GARD-FW, enter the following command in the Git Bash terminal:

```
$ git clone git@github.com:LSCC-Architecture/HUB.git GARD-FW
```

   Note: The URL is pasted from the clipboard copied in step 3.

7. After executing the command, you can see a folder named **GARD-FW** containing the complete repository. Navigate to this folder to work with the repository contents.

8. Checkout the branch or tag you need. For example, to checkout the code with tag *GARD-MOD-FW-v2.1.0*, enter:

```
$ git checkout GARD-MOD-FW-v2.1.0
```

   Note: Since GARD-MOD-FW-v2.1.0 is a tag, you will be in a *detached head* state. This is acceptable if you do not intend to contribute back to the source.

```
$ git checkout GARD-MOD-FW-v2.1.0
Note: switching to 'GARD-MOD-FW-v2.1.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at b625b0c Files Updated for 2.1.0 release
```

Figure A.8. Git Checkout

9. You now have the complete source code for tag *GARD-MOD-FW-v2.1.0* on your machine under the **GARD-FW** folder.

A.1.3. Building Firmware Image

The firmware root image that is written to the SPI NOR, consists of the following components:

- Firmware binary
- ML network binaries
- Camera configurations

The following subsections describe the process for building these binaries and placing them within the firmware root image.

A.1.3.1. Building Firmware Binary from Sources

After obtaining the firmware sources using one of the methods outlined in the Getting Firmware Source section, proceed to build the firmware code. Before starting the build process, ensure that all tools listed in the Development Environment section have been installed successfully. If the tools are not installed or if any errors occurred during installation, the firmware build will fail.

Building the firmware code also builds the firmware loader because both share common files. Follow these steps to generate the firmware binary:

1. Launch the Microsoft Visual Studio Code application.
2. From the toolbar, click **File**, then select **Open Workspace from File**.

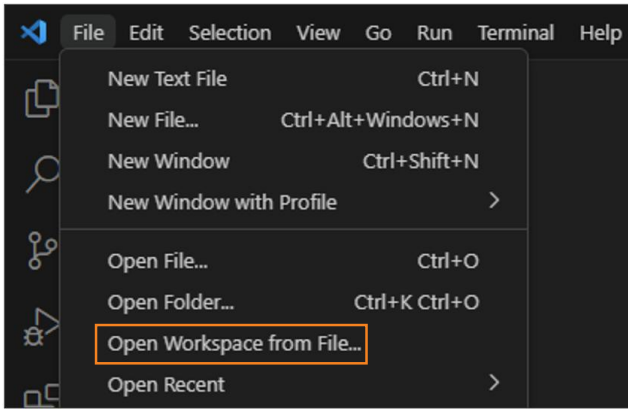

Figure A.9. Open Workspace from File

3. Navigate to the *GARD-FW/src/gard_firmware* folder and open the *gard_firmware.code-workspace* file.

4. This opens the GARD firmware workspace and lists all files and folders relevant for firmware development.

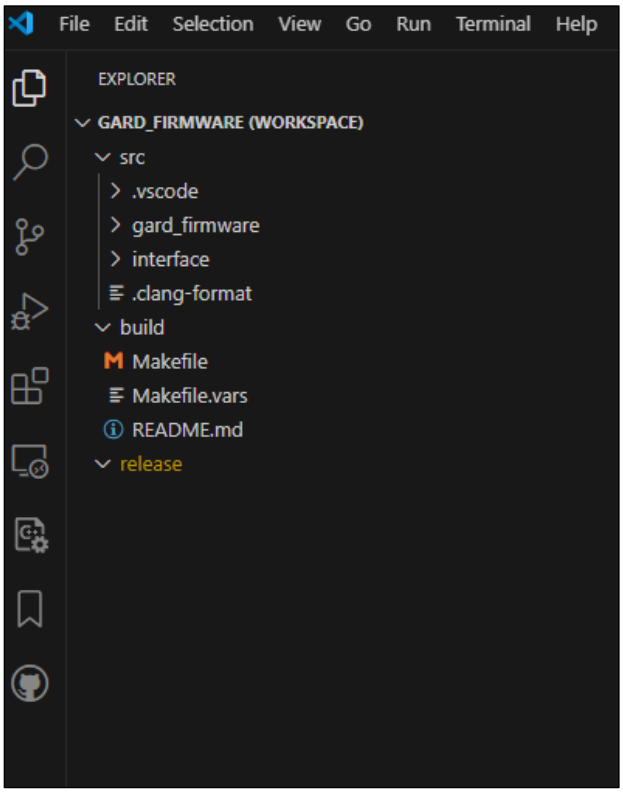

Figure A.10. Loaded Workspace View

5. Set the **Makefile Tools** extension to build the code by clicking the **Makefile** button on the left pane to view the current options for Makefile Tools.

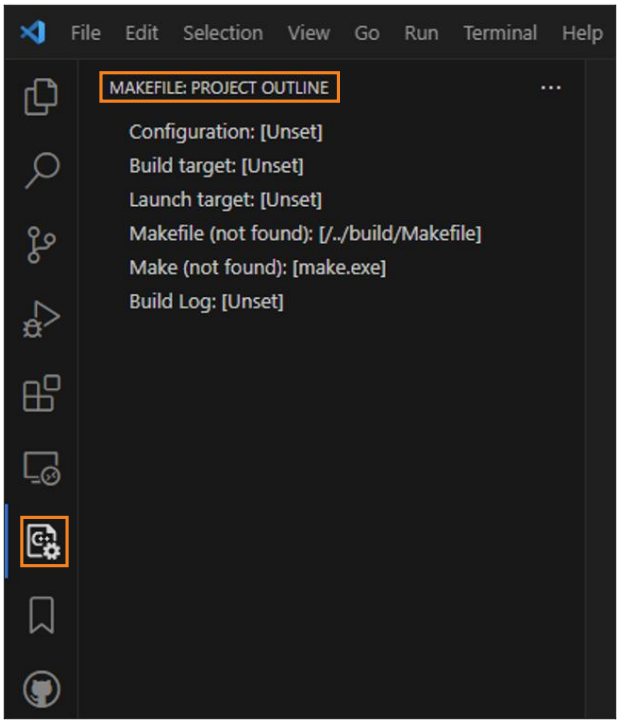

Figure A.11. Makefile Extension

6. Verify that the Makefile path points to the correct folder containing the Makefile.
7. Once the Makefile Tools extension is configured and the default target in the Makefile is set to build the firmware, the setup is complete.
8. From the toolbar, click **View**, then select **Command Palette**.

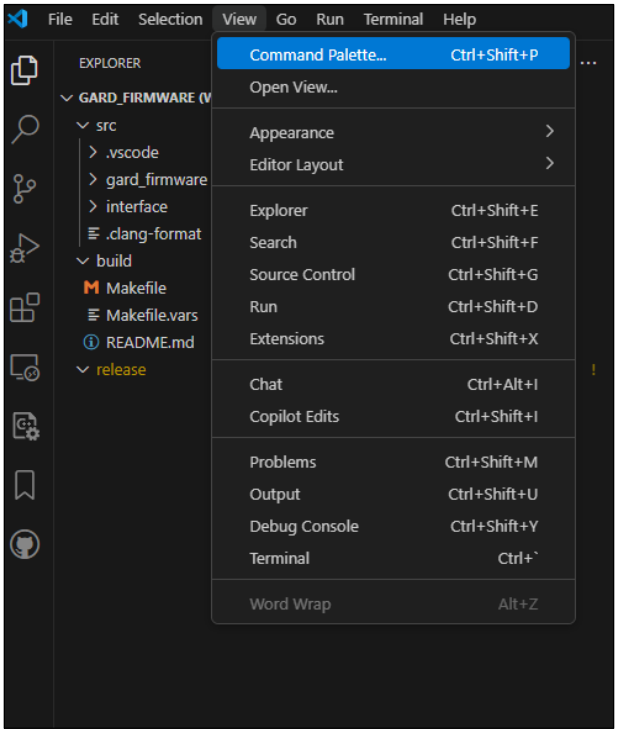

Figure A.12. Command Palette

9. In the Command Palette search box, enter *Clean and build*. Select **Makefile: Clean and build the current target**. The terminal opens and runs the build process to completion.

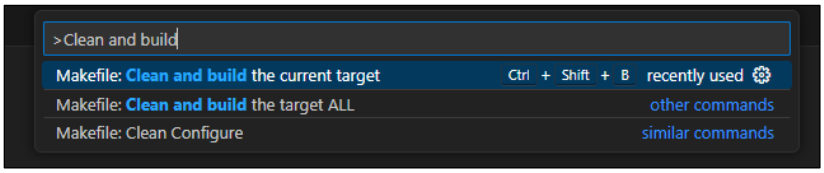

Figure A.13. Makefile Clean and Build

10. After the build completes, the firmware and firmware loader binaries are located in:
    - *build/output/gard_firmware/fw/gard_fw.bin*
    - *build/output/gard_firmware/fw/gard_fw.elf*
    - *build/output/gard_firmware/fw_ldr/gard_fw_ldr.bin*
    - *build/output/gard_firmware/fw_ldr/gard_fw_ldr.mem*
    - *build/output/gard_firmware/fw_ldr/gard_fw_ldr.elf*

A.1.3.2. Building ML Network Binary

The ML network binary must be compiled for the ML engine embedded in GARD. Obtaining the ML network file, compiling the network, and generating the final .bin file are beyond the scope of this document. This document assumes these steps are completed independently and that the ML network binary is available for use.

A.1.3.3. Building Camera Configuration

The camera connected to GARD must be configured to stream video in a format that mini ISP and ML engine can process. Currently, camera configurations are available for the Sony IMX 219 camera sensor in two aspect ratios: 4:3 AR and 16:9 AR. Since the same sensor can output images in either aspect ratio, separate configuration files are available for each.

There are several ways to obtain the camera configuration:

- Hand-code the configuration by consulting the camera sensor documentation.
- Capture an I2C kernel trace of actions executed by the kernel driver (provided by the camera vendor) during bootup.

You can use either method, but results may vary depending on the configuration and your requirements. In all cases, the camera configuration must be converted into a text format before it can be processed by the script that generates the binary.

A python script, *CameraConfigBuilder.py*, is provided in the (*HUB/src/scripts)* folder to build the camera configuration binary. The script takes a YAML input file containing configuration parameters to include in the binary.

The YAML file includes the following parameters:

- *Basic Config* – Specifies layout version, driver version, vendor name, model number, and supported interfaces (such as, I2C, MIPI, and GPIO).
- *Output File* – Name of the output camera configuration file.
- *Camera Configurations* – Provides the UID and along with *.bin* or *.txt* file generated from the I2C kernel trace.

Run the script using the following command:

```
$ python RootImageBuilder.py Camera_Config.yaml
```

A.1.3.4. Building Firmware Root Image

To consolidate all binaries into one firmware root image, use the *RootImageBuilder.py* Python script located in the HUB *HUB/src/scripts* folder. A YAML configuration file (*RFS-Config-For-MOD.yaml*) is provided. This file must contain the paths to the binaries that need to be included in the firmware root image.

The YAML file includes the following parameters:

- *Output File* – Specifies the name of the generated binary image file.

- *ML Networks* – Maps ML Model files to unique UIDs (0x2001 to 0x2FFF).
- *GARD Firmware* – Maps GARD firmware files to unique UIDs (0x1001 to 0x1FFF).
- Camera Configurations: Maps camera configuration files to unique UIDs (0x3001 to 0x3FFF).
- *App Data* – Maps user application data files to unique UIDs (0x5001 to 0x5FFF).
- *RFS Configuration* – Defines the RFS layout parameters (such as, layout version, update count, control fields, and boot firmware UID).
- *Flash Erase Block Alignment* – Minimum block size for flash erase operations (default: 0x8000 = 32 kB)
- *RFS Dir Copies* – Number of directory copies to maintain for redundancy.

Note: RFS configuration parameters define critical layout and boot settings. Incorrect values may result in an unusable image file. It is recommended to use default values unless specific requirements exist.

To generate the firmware root image, run the following command:

```
$ python RootImageBuilder.py -n -y RFS-Config-For-MOD.yaml
```

Ensure all binary files are accessible relative to the script's execution directory. The script generates a structured binary image using a Root File System (RFS) layout compatible with GARD.

A.1.4. API Documentation

GARD firmware sources are categorized into the following components:

- Firmware core: API defined in the *src\gard_firmware\inc\fw_core.h* file.
- Application module: API defined in the *src\gard_firmware\inc\app_module.h* file.

A.1.4.1. Firmware Core

APIs for the firmware core are exposed in the *src\gard_firmware\inc\fw_core.h* file. This file defines interfaces intended for use by the application module. These APIs provide a consistent, hardware-agnostic way for the application module to interact with GARD hardware. By abstracting hardware-specific details, this interface layer ensures that application logic can be reused across different hardware implementations without modification. This design promotes modularity, portability, and ease of maintenance across GARD platforms.

Structure, macros, and key functions for the firmware core are described in the following sections.

struct app_module_cbs_table

This structure defines the list of callbacks that are implemented by the application module. These callbacks are invoked by the firmware core at various stages of the application flow.

In XIP mode, the firmware loader relocates the addresses in this structure to the actual XIP location of these callbacks on AXI bus. The application module uses the *DEFINE_APP_MODULE_CALLBACKS()* macro to create a variable of this type and populate it with the application module callback functions.

#define DEFINE_APP_MODULE_CALLBACKS(preinit_fn, init_fn, preprocess_fn, ml_done_fn, img_proc_done_fn, rescale_done_fn)

The application module calls this macro outside of any function in one of its C source file to define the callbacks implemented by the application module. This macro creates a variable of type *struct app_module_cbs_table* in a specific memory section.

Ensure that only one C source file calls this macro to avoid multiple definitions, which can cause a link-time error. This macro is used by the application module to define the relocation table for XIP mode and must be called globally in any one of the application module C source file. This macro creates a variable that is accessed by the firmware core as well.

bool set_uart_parameters(void *uart_handle, uint32_t baud_rate, uint8_t parity, uint8_t stop_bits)

The application module calls this function to configure UART parameters such as baud rate and parity. Call this function only from within the *app_preinit()* function of the application module.

bool set_i2c_target_parameters(void *i2c_handle, uint32_t clock_speed, uint8_t target_address)

The application module calls this function to configure I2C parameters such as clock speed and target address. Call this function only from within the *app_preinit()* function of the application module.

bool register_networks(struct networks *list_of_networks)

The application module calls this function to register its ML networks with the firmware core. Call this function exclusively from the *app_init()* routine of the application module. After calling, the firmware core owns the networks variable, and the application module must not modify its contents.

bool schedule_network_to_run(ml_network_handle_t network)

The application module calls this function to schedule the next ML network to run on the ML engine. If not called, the first registered network runs by default. The parameter network specifies the UID of the already registered network that runs next on the ML engine.

ml_network_handle_t get_uid_of_next_network_to_run(void)

The application module calls this function to return the UID of the next ML network scheduled to run. Returns *INVALID_NETWORK_HANDLE* if no networks are registered.

int32_t get_uid_of_currently_running_network(void)

The application module calls this function to return the UID of the currently running ML network. Returns *INVALID_NETWORK_HANDLE* if none are running.

void *register_buffer_for_host_data(uint8_t *buffer, uint32_t buffer_size, rx_handler_t app_rx_handler)

The application module calls this function to register a buffer for receiving data from the application module counterpart running on the host. The *rx_handler_t* callback is invoked when data is received. The application module implements this handler to process incoming data.

void stream_data_to_host_async(uint8_t *data, uint32_t count_of_data_bytes, uint32_t timeout_ms, uint8_t *p_send_complete)

The application module calls this function to send streaming and recurring data, such as information generated by the pipeline, to the host. The data is sent asynchronously, meaning the transfer can complete in the background. The application module populates the buffer only when the **p_send_complete* parameter is set to true. The application module can use two buffers to accumulate new data while the transfer completes.

bool send_event_to_host(uint8_t *event_data, uint32_t count_of_data_bytes, uint32_t timeout_ms)

The application module calls this function to send asynchronous data to the host. This function is used to transmit unexpected events, such as errors or notifications that are not part of the regular command-response flow.

void capture_image_async(void)

The application module calls this function to initiate image capture from the connected camera. Once the image is captured, the firmware core calls the *app_preprocess()* function, allowing the application module to perform any preprocessing on the image data before the data is sent to the ML engine for processing.

void start_ml_engine(void)

The application module calls this function to start the ML engine. Call this function after the application module registers its networks and is ready to process images. The ML engine starts processing the image data and triggers an interrupt when processing is complete.

bool crop_and_rescale_image(struct image_info *in_image, struct image_info *scaled_image)

The application module calls this function to crop and rescale image data before sending the data to the ML engine for processing. The *in_image* parameter contains the original image data, while the *scaled_image* parameter contains the cropped and rescaled image data. This routine initiates a hardware operation to rescale the image, which may take time.

void schedule_image_processing_done_event(void)

The application module calls this function to schedule a callback from the firmware core after all pending events have been serviced. The firmware core then calls the *app_image_processing_done()* function, which the application module must implement to perform final processing after image processing is complete.

uint32_t read_module_data(uint32_t module_uid, uint32_t module_read_offset, uint32_t read_bytes, uint8_t *buffer)

The application module calls this function to read data from the module into the provided buffer, starting at the specified offset and reading the specified number of bytes.

A.1.4.2. Application Module

The API for the application module is exposed by the *src\gard_firmware\inc\app_module.h* file. This file defines the interfaces intended for use by the firmware core.

The functions listed in this file are implemented by the application module. These prototypes ensure that the firmware core can call these functions with the correct parameters.

app_handle_t app_preinit(void)

The firmware core calls this function before the main application initialization. The function provides the application module an opportunity to perform pre-initialization tasks, such as setting parameters that are used by the firmware core during initialization. The function returns an application handle, which is passed to subsequent *app_ *()* routines for accessing the application module context.

app_handle_t app_init(app_handle_t app_context)

The firmware core calls this function to initialize the application module after the firmware core initializes hardware blocks and its internal structures. The application module uses this call to initialize its own structures and prepare for the application flow. If camera capture is required, the application module invokes *start_image_capture()* from within this function.

enum app_ret_code app_preprocess(app_handle_t app_context, void *image_data)

The firmware core calls this function after the captured image is available and before the ML engine begins processing. This function allows the application module to perform any required preprocessing on the image data using the provided application context and image buffer.

enum app_ret_code app_ml_done(app_handle_t app_context, void *ml_results)

The firmware core calls this function when the ML engine finishes processing the image data. The function provides the application module an opportunity to perform any post-processing on the ML results and advance application-specific logic based on inference outcomes.

enum app_ret_code app_image_processing_done(app_handle_t app_context)

The firmware core calls this function when the image processing operation completes. Any further actions, such as sending results to the host, are performed here. This function is invoked only when explicitly requested by the application module through the firmware core scheduling mechanism.

enum app_ret_code app_rescale_done(app_handle_t app_context)

The firmware core calls this function when the image rescaling operation completes. The function allows the application module to perform any actions needed after rescaling, such as updating state, scheduling subsequent steps, or preparing data for downstream processing.

A.1.5. Debugging Firmware

This section describes the steps required to set up and establish a working JTAG based debug connection between the development environment and the GARD target. Before starting the process, ensure that all tools listed in the Development Environment section have been installed successfully.

A.1.5.1. Prerequisites

Compiling Firmware Binaries

Follow the steps in the Building Firmware Binary from Sources section to generate firmware binaries. These binaries, particularly *build/output/gard_firmware/fw/gard_fw.elf* and *build/output/gard_firmware/fw_ldr/gard_fw_ldr.elf*, are required for debugging the firmware.

Flashing SPI NOR Flash

Flash the firmware root image onto SPI NOR Flash. These binaries, particularly the GARD bitstream and firmware root image, must be flashed onto SPI NOR Flash before debugging the firmware.

Powering Connection with Target SOM Board

Power the GARD-based target SOM board using specified power supply. Ensure a USB cable connection between target board and development workstation using the specified micro–USB A cable. Once connected, the Windows Device Manager utility displays the specified COM ports in ascending numerical sequence.

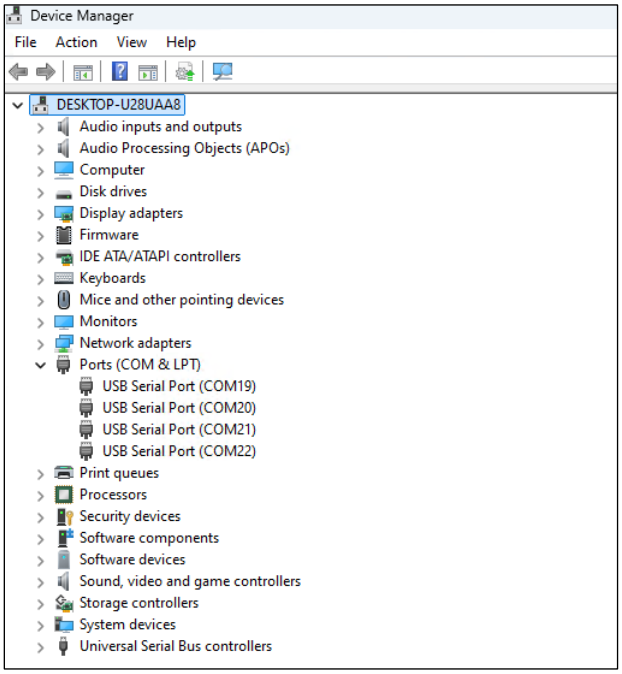

Figure A.14. COM Ports on GARD-Based SOM Board

A.1.5.2. Establishing JTAG-Based GDB Connection with Target SOM Board

Follow the steps below to establish a JTAG connection with the target SOM board.

Configuring GDB Scripts to Connect to RISC-V Processor Core

Upon power cycle, the firmware residing in *fw_root.bin* flashed onto SPI NOR Flash starts executing. To attach only to the running process on RISC-V core, modify the *tools\loadfw.gdb* file to prevent GDB from loading the *build\output\gard_firmware\fw\gard_fw.elf* firmware image onto TCM:

1. Locate the **load** command in the GDB script.

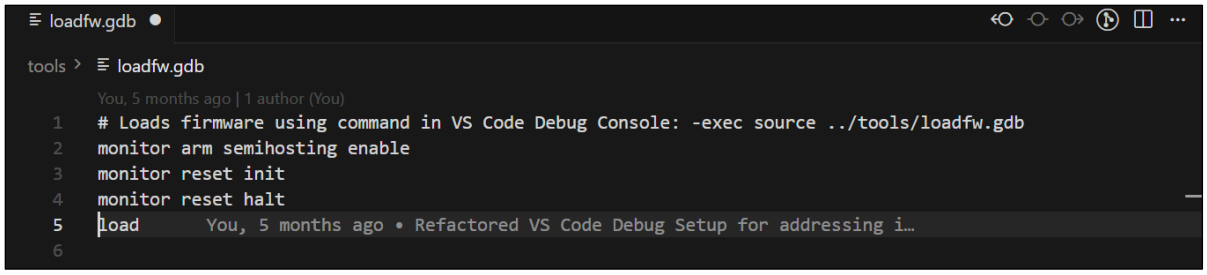

Figure A.15. Locate *load* in *build\output\gard_firmware\fw\gard_fw.elf*

2. Comment out the **load** command to prevent GDB from loading firmware onto TCM.

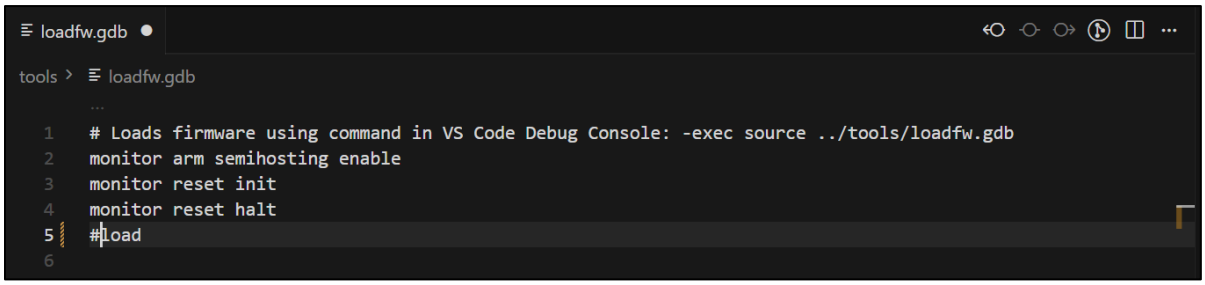

Figure A.16. Comment Out *load* in *build\output\gard_firmware\fw\gard_fw.elf*

Initiating GDB Debug Session

1. Navigate to the **Run and Debug** window in Microsoft Visual Studio Code.
2. Initiate the GDB connection to RISC-V firmware using *Attach Firmware to RISC-V OPEN OCD Server (Workspace)* by clicking the green play button.
3. Follow the same steps to initiate GDB connection to the RISC-V firmware loader but use the target: *Attach Firmware Loader to RISC-V OPEN OCD Server (Workspace)*.

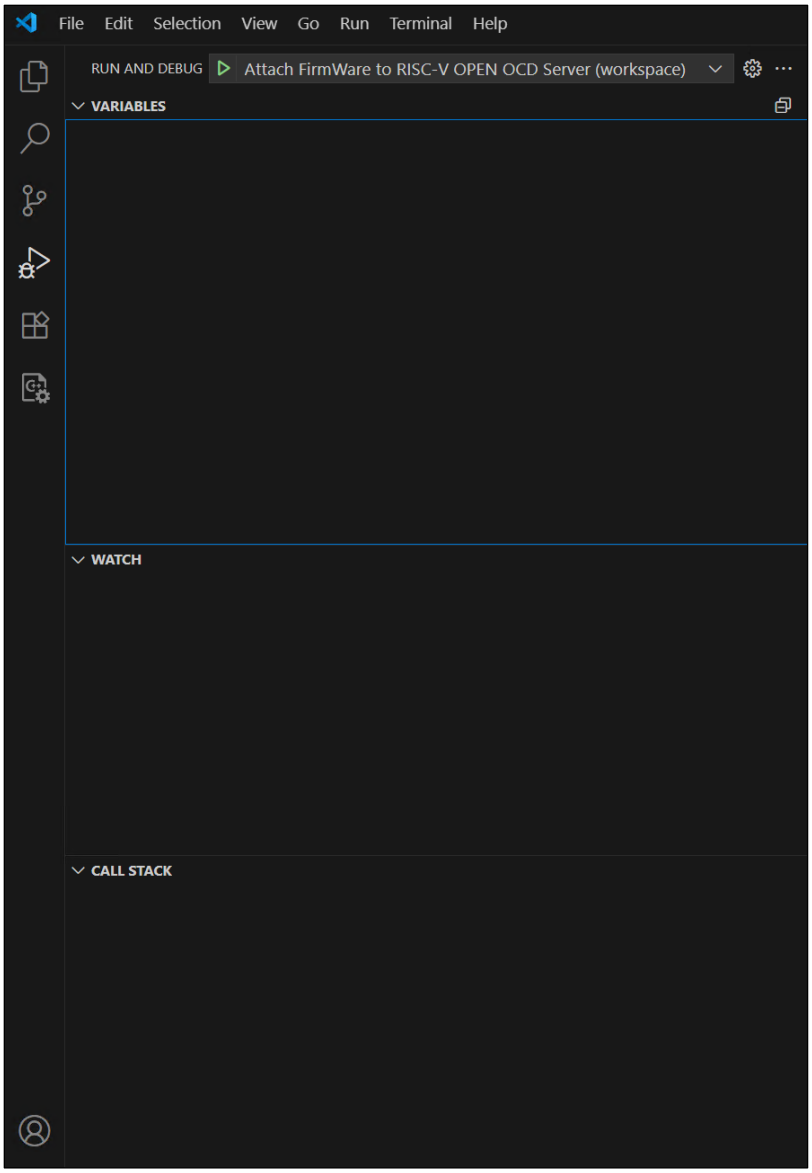

Figure A.17. Initiate GDB Session to Firmware

Attaching Microsoft Visual Studio Code GDB Client to Open OCD Server

Successful Open OCD execution displays a dialog box. Microsoft Visual Studio Code then attaches for source-level debugging.

Click **Debug Anyway** only after the *Info : Listening on port 3333 for gdb connections* log message appears in the console, which attaches the GDB client to the Open OCD server.

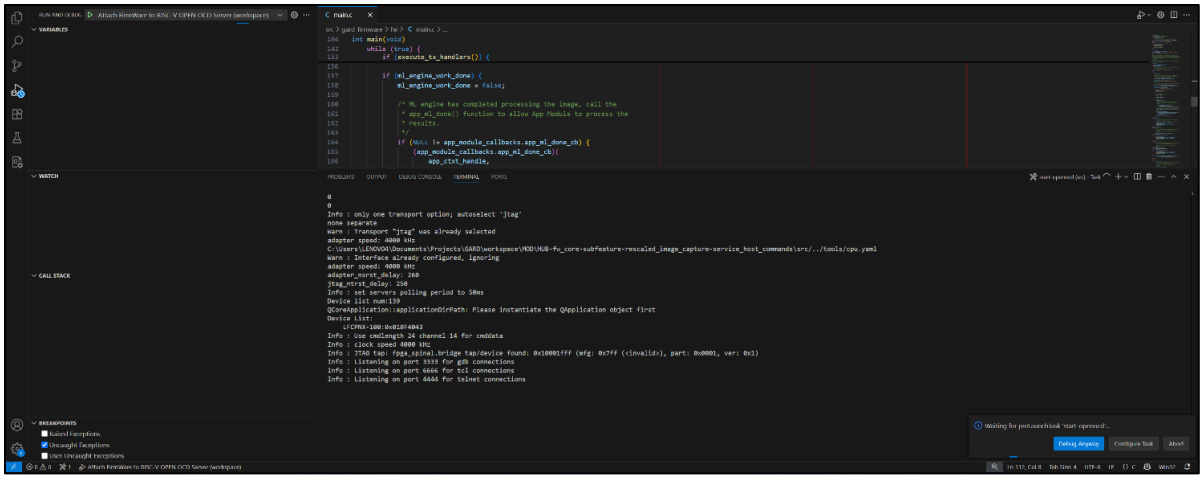

Figure A.18. Attach Using *Debug Anyway*

Debugging Firmware Sources

After successful attachment of Microsoft Visual Studio Code's GDB client to the GARD-based SOM target running RISC-V through the Open OCD server, the Debug UI appears in Microsoft Visual Studio Code.

Navigate to the **Debug Console** tab.

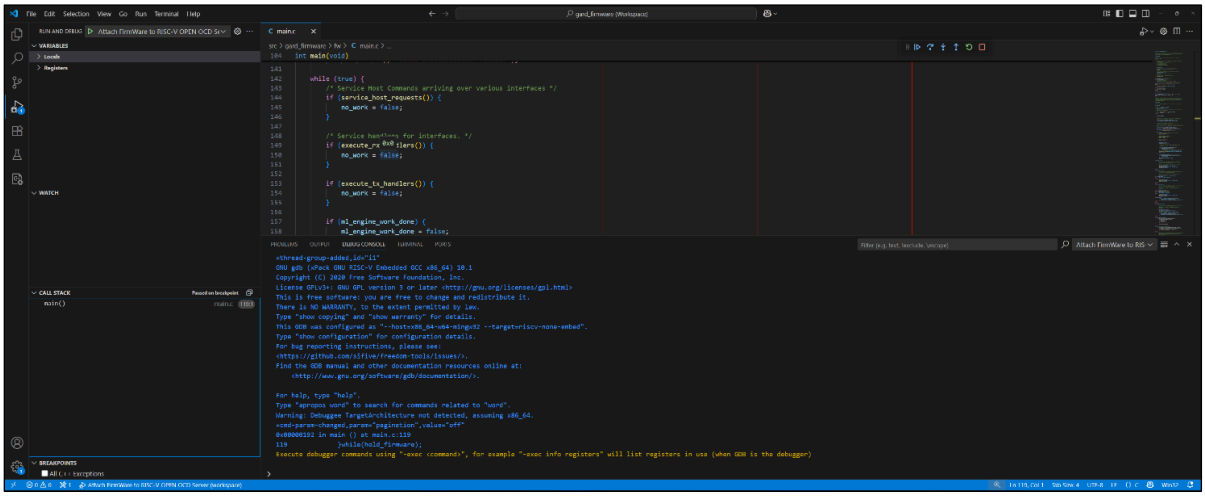

Figure A.19. Debug Console Tab in Microsoft Visual Studio Code

From this point onward, you can debug the firmware using the Microsoft Visual Studio Code debug interface. Common operations such as setting breakpoints and watching variables can be performed using the left pane of the debug window.

Microsoft Visual Studio Code also allows execution of GDB commands in **Debug Console**. Any GDB command can be executed using the following format:

```
-exec <GDB command>
```

Note: A GDB command must be prefixed with *-exec*.

For example, to inspect the current value of the program counter:

```
-exec p $pc
```

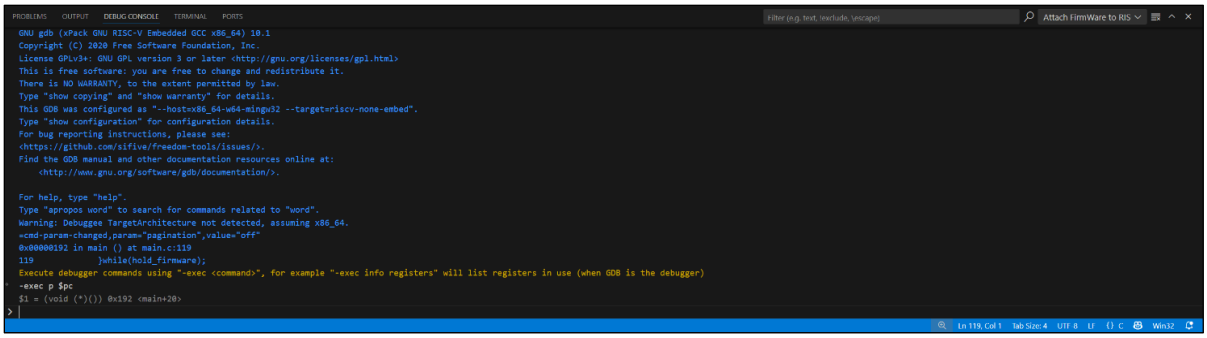

Figure A.20. Executing GDB Command in Debug Console

Terminating Debug Session

1. Click **Stop** (a square-shaped red button) to detach the GDB connection in Microsoft Visual Studio Code.

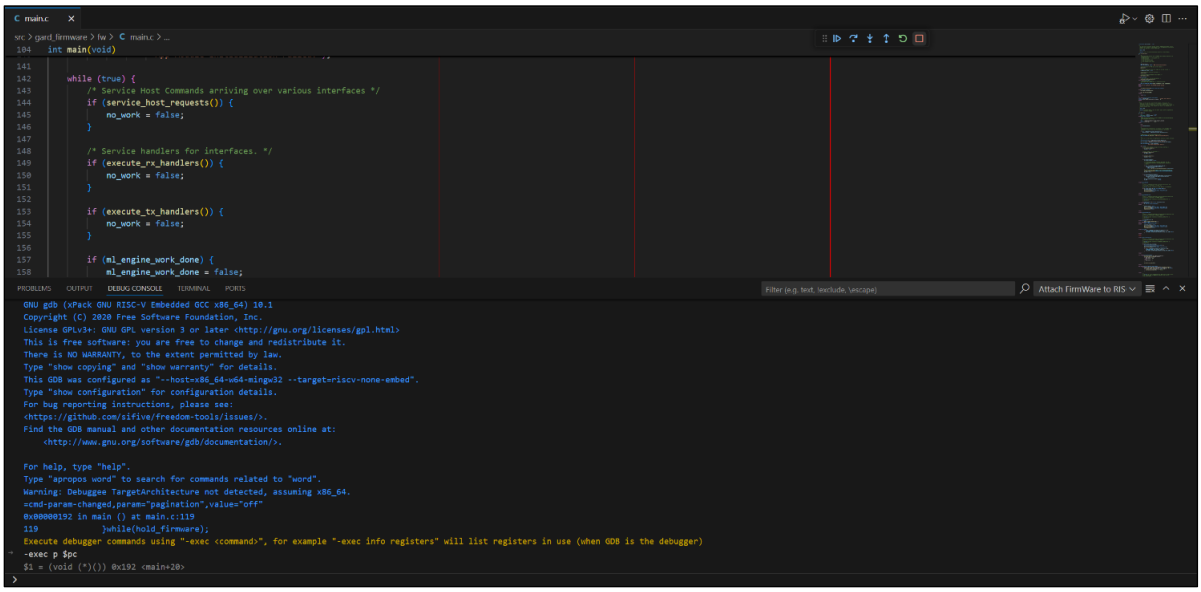

Figure A.21. Stop Debug Session

2. Navigate to the **Terminal** tab and press **Ctrl + C** to terminate the Open OCD server connected to the GARD-based SOM board running RISC-V.

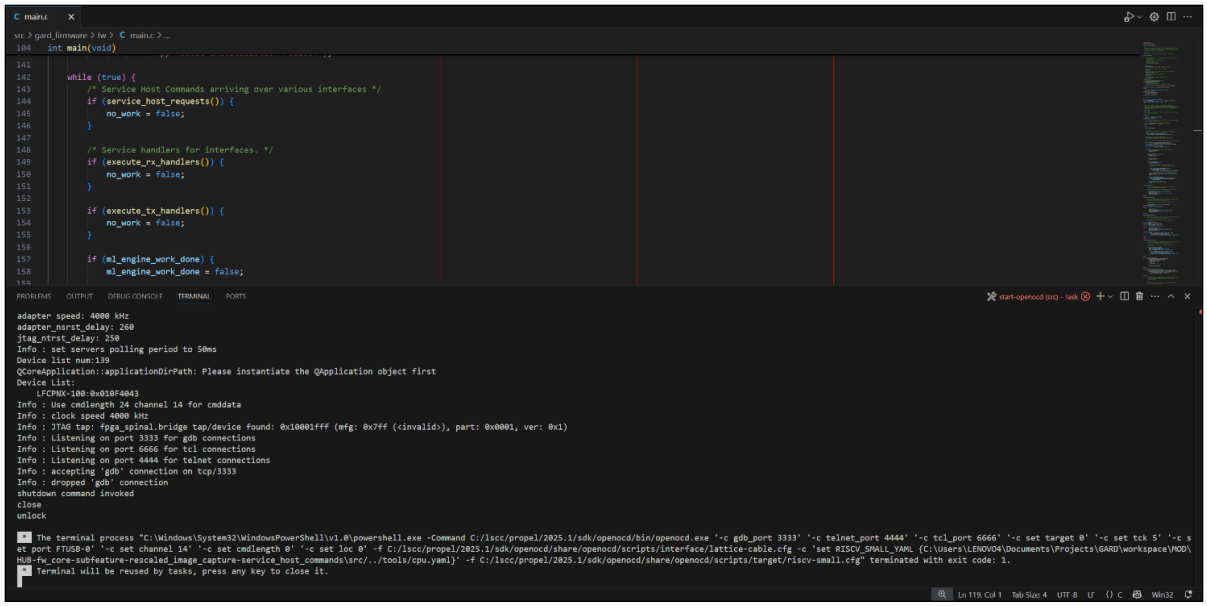

Figure A.22. Terminate Open OCD Server

3. Verify that GDB connection is successfully terminated by opening **Task Manager** in Windows and search for *cable server*. Ensure no process is running. If it is, terminate it manually.

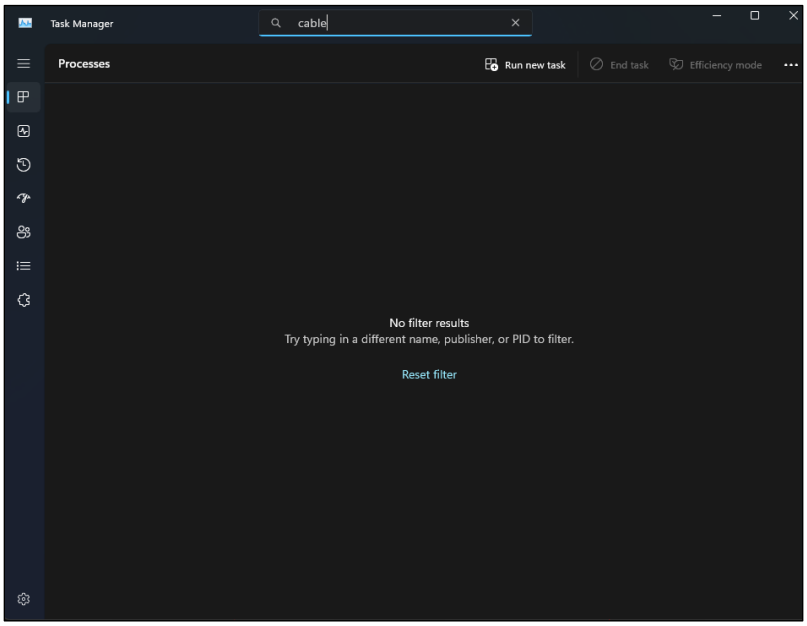

Figure A.23. Verify GDB Connection Successfully Terminated

References

- CertusPro-NX Family Data Sheet (FPGA-DS-02086)
- Advanced CNN Accelerator IP User Guide (FPGA-IPUG-02224)
- APB Interconnect Module User Guide (FPGA-IPUG-02054)
- AXI4 Interconnect Module User Guide (FPGA-IPUG-02196)
- AXI4 to APB Bridge Module User Guide (FPGA-IPUG-02198)
- GPIO IP User Guide (FPGA-IPUG-02076)
- I2C Controller IP User Guide (FPGA-IPUG-02071)
- I2C Target IP User Guide (FPGA-IPUG-02072)
- Octal SPI Controller IP User Guide (FPGA-IPUG-02273)
- RISC-V RX CPU IP User Guide (FPGA-IPUG-02298)
- SGDMA Controller IP Core User Guide (FPGA-IPUG-02131)
- System Memory Module User Guide (FPGA-IPUG-02073)
- Tightly-Coupled Memory IP User Guide (FPGA-IPUG-02299)
- Tri-Speed Ethernet IP User Guide (FPGA-IPUG-02084)
- UART IP User Guide (FPGA-IPUG-02105)
- Lattice Radiant Timing Constraints Methodology (FPGA-AN-02059)
- CertusPro-NX Multi-Object Detection on SOM Board Demo Guide (FPGA-UG-02247)
- CertusPro-NX Defect Detection on SOM Board Demo Guide (FPGA-UG-02248)
- Lattice Memory Mapped Interface and Lattice Interrupt Interface (FPGA-UG-02039)
- Lattice IP Packager 2025.1 (FPGA-UG-02236)
- Reveal User Guide for Radiant Software
- Lattice sensAI Solution Stack web page
- CertusPro-NX web page
- MachXO3D web page
- Lattice Diamond Software web page
- Lattice Propel Design Environment web page
- Lattice Radiant Software web page
- Lattice Solutions IP Cores web page
- Lattice Solutions Reference Designs web page
- Lattice Insights web page for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

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

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

Revision History

Revision 1.0, January 2026

Section	Change Summary
All	Initial release.

www.latticesemi.com