Lattice NXP PCIe Companion User Guide

Reference Design

FPGA-RD-02343-1.0

September 2026

Disclaimers

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

Inclusive Language

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

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Acronyms/Abbreviations	Definition
AHB/AHBL	Advanced High-performance Bus/Advanced High-performance Bus Lite
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
CRAM	Configuration Random Access Memory
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DTB	Device Tree Blob
FPGA	Field-Programmable Gate Array
GPIO	General Purpose Input/Output
LMMI	Lattice Modular Memory Interface
LSB	Least Significant Bit
LUT	Look-Up Table
MSB	Most Significant Bit
OSPI	Octal Serial Peripheral Interface
PCIe	Peripheral Component Interconnect Express
SPI	Serial Peripheral Interface
USB	Universal Serial Bus

1. Introduction

The Lattice NXP PCIe® Companion reference design is aimed at providing an efficient methodology for updating FPGA bitstreams through the PCIe interface. This implementation incorporates a RISC-V embedded system integrated with the PCIe Full-Bridge IP. In this workflow, the host system transmits bitstream data across the PCIe bus, while the RISC-V soft processor manages the programming of the received data into the external SPI flash memory of the FPGA.

1.1. Quick Facts

The reference design files are available for download from the Lattice reference design web page.

Table 1.1. Summary of the Reference Design

General	Target Devices	LFCPNX-100-9LFG672C, LFD2NX-40-9BG256C
	Source code format	Verilog, C
Simulation	Functional simulation	Not performed
	Timing simulation	Not performed
	Test bench	Not available
	Test bench format	—
Software Requirements	Software tool and version	• Lattice Radiant™ Software 2026.1 • Lattice Propel™ Builder 2026.1
	IP version (if applicable)	• PCIe IP x1 v3.1.0.01/PCIE IP x4 v4.1.0.0.1 • RISC-V MC v2.9.0 • System Memory v2.6.0 • UART v1.6.0 • GPIO v1.8.1 • Octal SPI Controller IP v1.4.1 • AXI4 Interconnect v2.3.0 • APB Interconnect v1.5.0 • AHB-Lite to AXI4 bridge v1.5.0 • AXI4 to AHB-Lite bridge v1.6.0 • AXI4 to APB bridge v1.5.0 • FPGA Config v1.0.0 • PCIe LMMI App v.1.0.0 • PCIe System Memory Bridge v.1.0.0
Hardware Requirements	Board	NXP i.MX95 (Kernel 6.12.49-lts-next-gdf24f9428e38)
	Cable	• USB-A to USB-C cable • USB-A to Mini USB • NXP i.MX95 Power Supply • Lattice FPGA Power Supply

1.1.1. Quick Start Guide

The procedures for setting up the hardware and running the demonstration designs are described in the Running the Reference Design section.

1.2. Features

The key features of the FPGA PCIe Companion reference design include:

- High-Speed Data Retrieval – Enables the FPGA to initiate data reads from the host DMA region through the PCIe interface.
- Direct Memory Access – Supports FPGA-initiated data writes to the host DMA region for efficient data offloading.
- Bidirectional Host Transactions – Facilitates seamless host-driven memory read and write operations to the FPGA.
- In-System Programming – Provides the capability for the host to perform remote FPGA bitstream updates over the PCIe link.

1.3. Limitations

This reference design has the following known limitations:

- PCIe configuration space read and write are not supported.
- Legacy I/O read and write are not supported.

2. Directory Structure and Files

Figure 2.1 illustrates the directory structure.

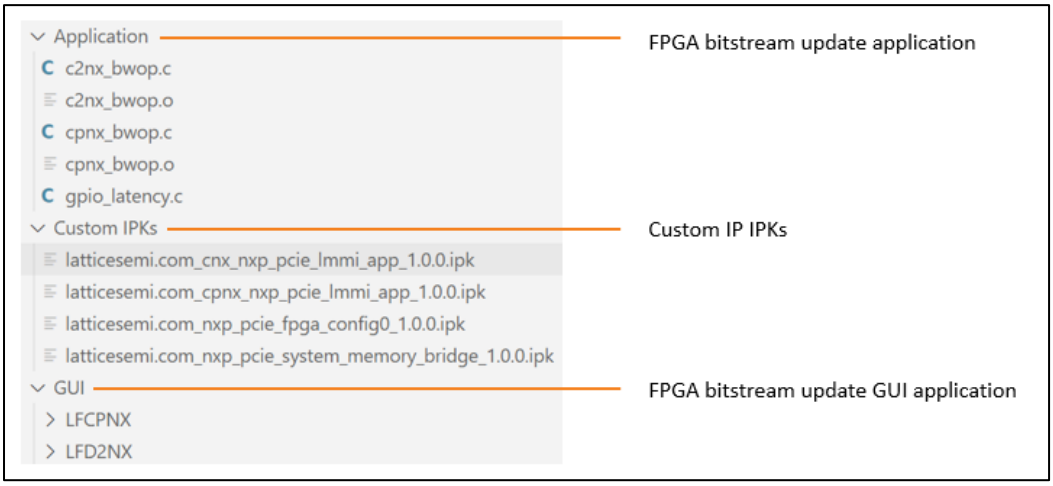


Figure 2.1. Directory Structure

Table 2.1. File List

Folder/File	Description
c2nx_bwop.c c2nx_bwop.o cpnx_bwop.c cpnx_bwop.o	Host application that is responsible for receiving input from the GUI and transmitting bitstream data to the FPGA over the PCIe interface. The .o files are the compiled binary outputs generated from their respective source code files.
gpio_latency.c gpio_latency.o	An application that measures PCIe transaction latency by monitoring the elapsed time between GPIO toggle events.
PcieFpgaUpdateGUI.exe	Windows-based graphical application that provides a user-friendly interface for initiating and managing FPGA bitstream updates.

3. Functional Description

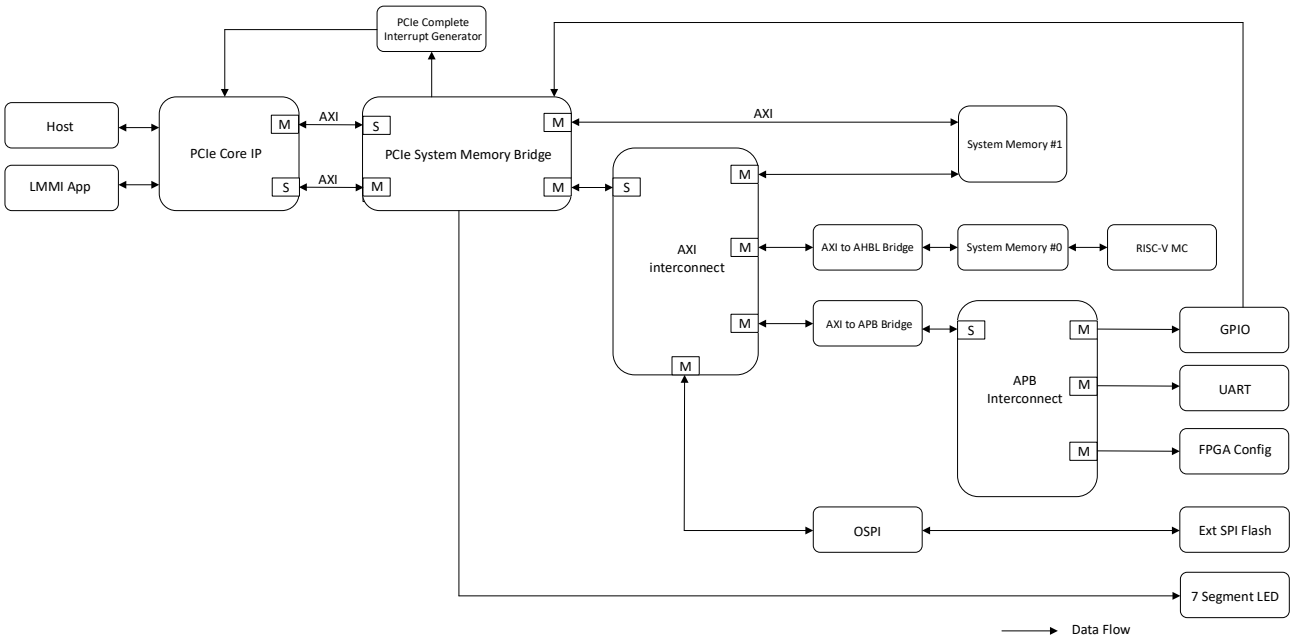


Figure 3.1. Reference Design Block Diagram

3.1. Design Components

The reference design includes the following main components:

Table 3.1. Main Components Overview

Component	Description
FPGA Configuration Module	Triggers FPGA reconfiguration by loading the updated bitstream from SPI flash into the configuration SRAM.
PCIe Complete Interrupt Generator	A module that triggers the PCIe Endpoint IP to send an interrupt to the host upon completion of the FPGA operation.
GPIO IP	Controlled by the RISC-V processor to signal the PCIe System Memory Bridge to copy data and return status information to the host PCIe Root Complex.
LMMI app Module	Initializes the PCIe configuration at boot-up prior to operation.
Octal SPI	Provides an interface to the external SPI flash device.
PCIe Core IP	Provides PCIe endpoint functionality within the FPGA.
PCIe System Memory Bridge	Connects PCIe interface to system memory while also functioning as an autonomous DMA engine. It monitors writes to a mailbox register set. Upon a valid doorbell trigger, autonomously executes bulk data transfers in either direction between PCIe host memory and system memory. Upon completion, it asserts a status signal to the PCIe interface. It also exposes two profiling outputs to measure transfer latency in each direction. More detailed information is explained in a later section.
RISC-V	Executes the application code responsible for handling PCIe data and FPGA configuration updates.
System Memory #0	Embedded memory used to execute the RISC-V application.
System Memory #1	Serves as a buffer for data received through PCIe.
UART	Outputs diagnostic and debug messages from the RISC-V processor to an external terminal.

3.2. LMMI App Module

The LMMI App is present in the architecture for both the CertusPro™-NX and Certus™-NX devices. Its purpose is to configure the Lattice PCIe IP core through a sequence of LMMI register accesses prior to operation.

Table 3.2. Certus-NX LMMI Application

Operation	Register	Value	Purpose
Write	main_ctrl_1 (0x0F004)	0x0001_0001	Puts the PCIe Link Layer core into a held reset state before any configuration is applied. Without hold_reset, the reset would self-clear after one clock cycle and configuration writes would race against a live core.
Write	pm_aspm_l0s (0x03040)	0x0000_0000	Explicitly disables ASPM L0s entry. The reset default for the enable bit is 1 (enabled), so this write is necessary to prevent the power-management state machine from autonomously entering the L0s low-power state during the initialization window before link training has completed.
Write	pm_aspm_l1 (0x03050)	0x0000_0000	Explicitly disables ASPM L1 entry. Like L0s, the reset default is enabled. Disabling L1 prevents a deeper power-saving state from being entered autonomously during initialization, which would stall or abort the LTSSM link training sequence.
Write	pm_aspm_l1_min (0x03054)	0x0000_0000	Sets the minimum time between ASPM L1 re-entry requests. Writing zero to both fields activates the PCIe specification default of 9,500 ns for the re-entry interval while keeping enforcement enabled. This ensures L1 is not requested again too rapidly once it is eventually re-enabled after link training.
Read	stat_port_0 (0x0F200)	—	Polls the TX PLL lock status from the PMA interface. The core must remain in reset until the SERDES TX PLL has fully acquired lock; releasing reset while the PLL is still settling would result in an unstable reference clock and corrupt LTSSM link training. The state machine loops on this read until bit[4] goes high.
Write	main_ctrl_1 (0x0F004)	0x0000_0000	Releases the PCIe Link Layer core from reset now that the PLL is stable and all power-management registers are configured. The core immediately begins LTSSM link training. config_done is asserted after this write completes.

Table 3.3. CertusPro-NX PCIe Bridge Board LMMI Application

Operation	Register	Value	Purpose
Write	main_ctrl_1 (0x0F004)	0x0001_0001	Puts the PCIe Link Layer core into a held reset state before any configuration is applied. Without hold_reset, the reset would self-clear after one clock cycle and configuration writes would race against a live core.
Write	msi_cap (0x040E8)	0x0000_0032	Configures the MSI capability block that is visible to the host during PCIe enumeration. Requesting 8 vectors (encoding 3) and enabling per-vector masking gives the driver flexibility to assign individual interrupt sources to distinct vectors and mask them selectively.
Write	Interrupt (0x04050)	0x0000_0000	Keeps legacy INTx interrupt support active and selects INTA as the interrupt pin. Since MSI is also enabled, the driver can choose whichever mechanism it prefers; INTA serves as the fallback for hosts or drivers that do not use MSI.
Read	PHY PMAstatus (0x007F)	—	Polls lane 1 TX PLL lock status through the per-lane PHY PMA status register. The core must not be released from reset until the SERDES TX PLL is stable; doing so while the PLL is still acquiring lock corrupts link training. The loop repeats until bit[4] of the 64-bit read data (bit[36] offset into the two-lane word) goes high.
Write	main_ctrl_1 (0x0F004)	0x0000_0000	Releases the PCIe Link Layer core from reset now that the PLL is confirmed stable and all configuration registers are programmed. The core begins LTSSM link training immediately. config_done is asserted after this write completes.

3.3. FPGA Configuration Module

The FPGA Configuration is used to trigger an internal FPGA REFRESH/PROGRAMN command to LMMI logic. This core IP implements an APB endpoint which decodes the RISC-V CPU command data. The LMMI host FSM inside is used to execute the soft reset to reload the bitstream from the physical SPI flash onto the FPGA CRAM.

3.4. Mailbox Architecture

The Mailbox serves as a handshake mechanism that coordinates communication between the i.MX95 host application and the FPGA RISC-V application.

Table 3.4. Mailbox Register

Name	Offset	Description
Command	0x00	The operation that the host directs the FPGA to execute.
Destination lower address	0x04	The host memory address to which upstream data is written.
Destination upper address	0x08	
Source lower address	0x0C	The host memory address from which the FPGA retrieves the previously written data.
Source upper address	0x10	
Trigger	0x14	0x1 = Signals the FPGA RISC-V application to begin processing the action specified in the Command register. 0x2 = Instructs the FPGA to copy data from the specified source address in host memory to the destination address in FPGA memory, with the transfer length defined in the data length register.
Flash start address	0x18	The target offset within the physical SPI flash where the 4KkB bitstream data block is written.
Data length	0x1C	The number of bytes to be written or copied.
Operation status	0x100	Indicates the result of the most recently executed FPGA operation.
Flash data	0x200	The FPGA-side buffer holding the 4 kB bitstream block prior to its transfer to the physical SPI flash.

Table 3.5 lists the supported commands.

Table 3.5. Mailbox Supported Commands

Command	Description
0x1	Instructs the FPGA to program the received 4 kB bitstream data to the physical SPI flash.
0x2	Instructs the FPGA to initiate a hardware reconfiguration sequence.
0x3	Instructs the FPGA to send the received 4 kB data back to the host.

Table 3.6 lists the defined operation status codes.

Table 3.6. Mailbox Operation Status

Status	Description
0x0	Unknown status (Default)
0x1	Fail
0x2	Success

3.5. Bitstream Update Flow

The flowchart below describes how a host system transfers a bitstream to an FPGA incrementally in 4 kB chunks, writing each chunk into flash memory using a doorbell-and-interrupt mechanism.

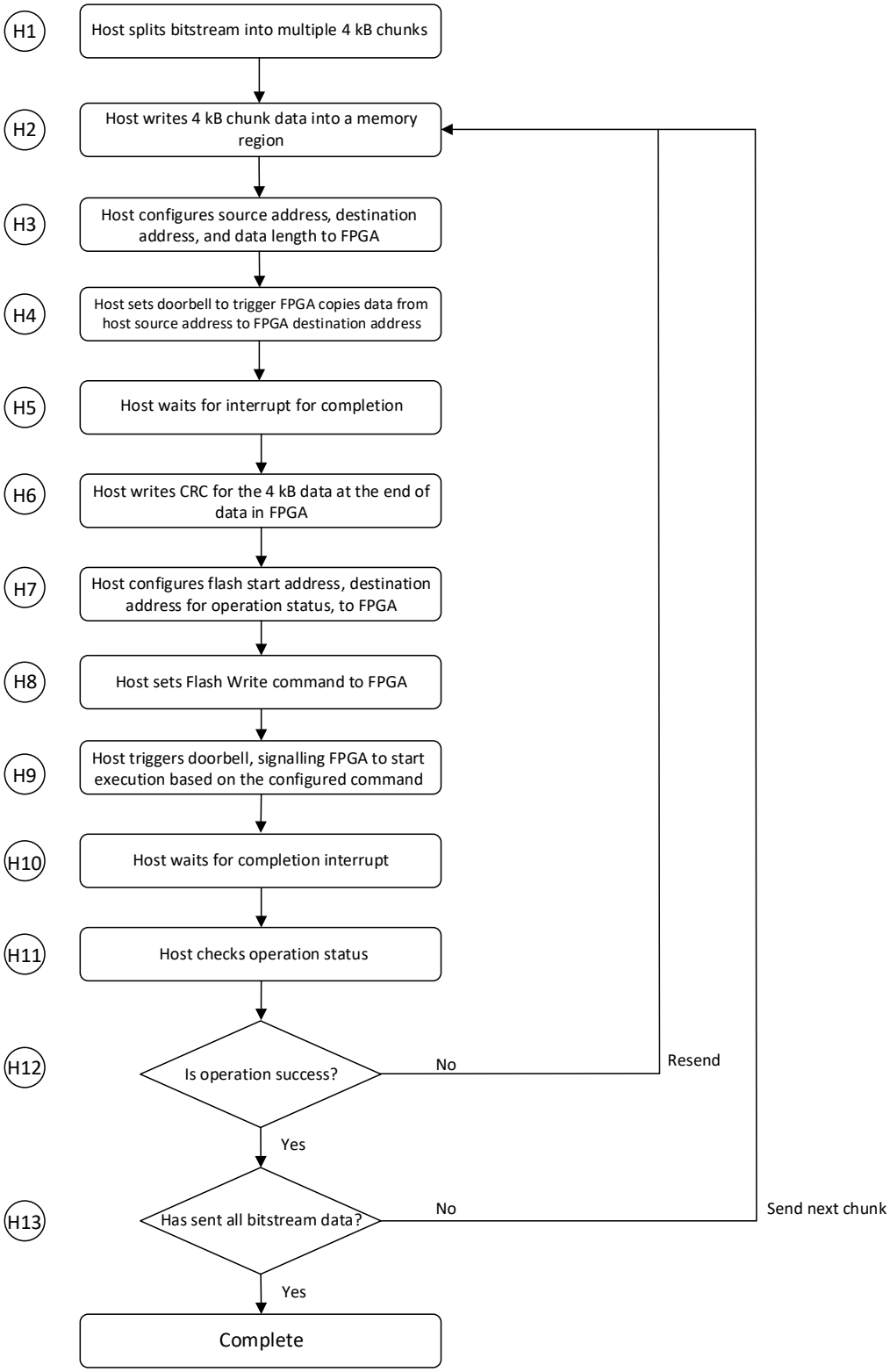


Figure 3.2. Flowchart of Host-Initiated FPGA Bitstream Update through the PCIe Interface

The process begins at *H1*, where the host splits the full bitstream file into multiple 4 kB chunks in preparation for transfer. The host then enters the main update loop at *H2* by writing one 4 kB chunk into a designated memory region. At *H3*, it configures the source address, destination address, and data length in the FPGA registers, providing the FPGA with the information it needs to locate and receive the data. The host then triggers a DMA transfer at *H4* by writing to the doorbell register, instructing the FPGA to autonomously copy the data from host memory into its own system memory. The FPGA-side transfer process is described in the FPGA Memory Read Flow section. The host pauses at *H5*, waiting for a completion interrupt from the FPGA before proceeding.

Once the transfer is confirmed, the host moves to *H6* and appends the CRC checksum for that 4 kB chunk to the FPGA data area, allowing the FPGA to verify the integrity of the received data. At *H7*, the host configures the target SPI flash address and the destination address for the operation status register. It then issues the Flash Write command to the FPGA at *H8*, and at *H9*, triggers execution by writing to the doorbell register a second time. The FPGA RISC-V processor then verifies the CRC, writes the data to SPI flash, and reports the result back to the host as described in the RISC-V Updates Bitstream Flow and FPGA Memory Write Flow sections. The host waits at *H10* for a completion interrupt confirming the flash write has finished.

At *H11*, the host reads the operation status register to determine the outcome. The flow then reaches the first decision point at *H12* — if the operation was unsuccessful, the flow loops back to *H2* to retry the same chunk. If the operation was successful, the flow advances to *H13*, where it checks whether all chunks have been sent. If more chunks remain, the flow returns to *H2* to begin processing the next 4 kB chunk. Once all chunks have been successfully written to SPI flash, the flow exits the loop and reaches the *Complet***e** state, indicating that the bitstream update has finished successfully.

3.6. FPGA Memory Read Flow

The flowchart below illustrates how the PCIe System Memory Bridge transfers data from the host into the FPGA's System Memory. This process is triggered when the host sets the doorbell register at step H4 of Figure 3.2, instructing the FPGA to autonomously fetch the data over the PCIe interface.

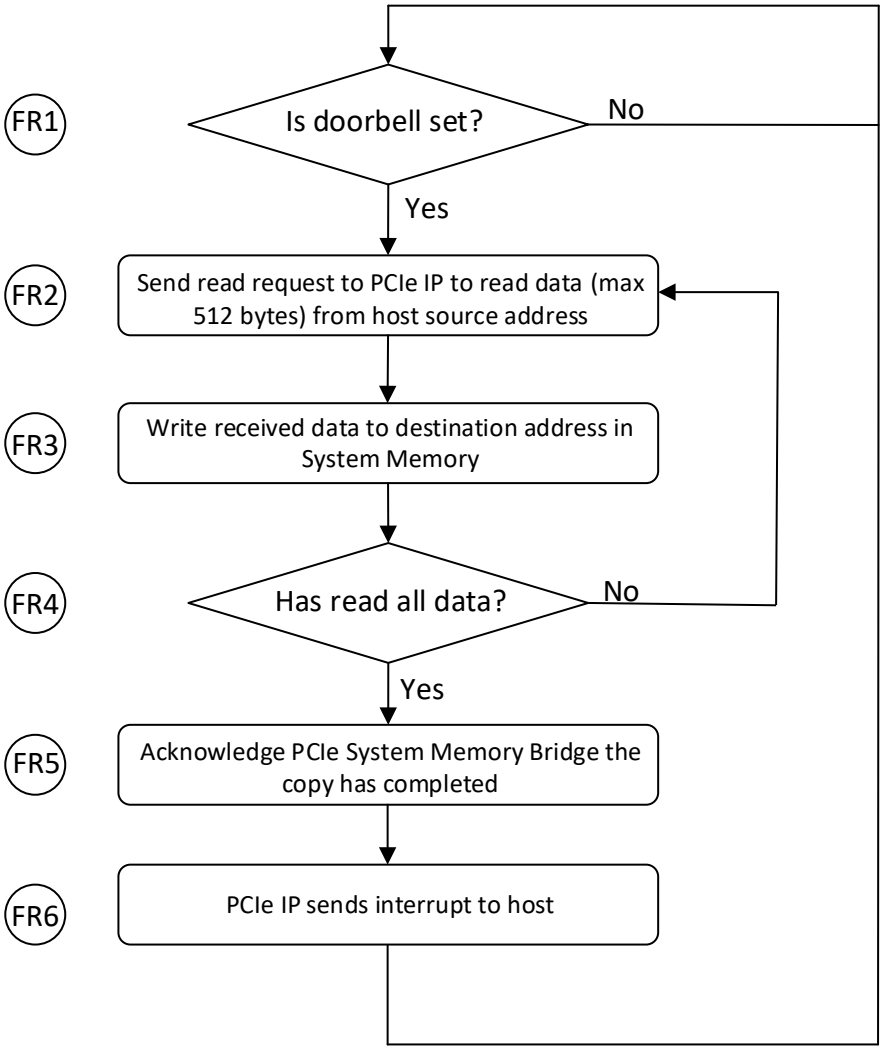


Figure 3.3. Flowchart of PCIe System Memory Bridge Data Transfer from Host to System Memory

The flow begins at *FR1* with a polling decision — the FPGA continuously checks whether the doorbell has been set. If the doorbell has not been set, the flow loops back and keeps checking. Once the doorbell is detected as set, meaning the Host has signaled that a read operation should begin, the flow proceeds forward.

At *FR2*, the FPGA sends a read request to the PCIe IP to fetch data of up to 512 bytes at a time from the host source address over the PCIe bus. This 512-byte limit is determined by the Maximum Read Request Size (MRRS) of the NXP i.MX95, which is set to 512 bytes. Once that data is received, the flow moves to *FR3*, where the FPGA writes the received data to the designated destination address in System Memory, effectively landing the data where it needs to reside.

After writing, the flow reaches the decision point at *FR4*, which asks whether all the required data has been read. If not all data has been read yet, the flow loops back to *FR2* to send another read request and fetch the next batch of up to 512 bytes. This loop between FR2, FR3, and FR4 continues, chunk by chunk, until all the data has been fully read and written into System Memory.

Once all data has been read, the flow exits the loop and proceeds to *FR5*, where the FPGA acknowledges to the PCIe System Memory Bridge that the copy operation is completed. Finally, at *FR6*, the PCIe IP sends an interrupt to the Host, notifying it that the entire memory read operation is done.

3.7. RISC-V Updates Bitstream Flow

The flowchart below illustrates how the RISC-V processor writes the incoming bitstream data into the external SPI flash. This process begins after the host sets the doorbell register at step H9 of Figure 3.2, signaling the RISC-V application to commit the received data to persistent storage.

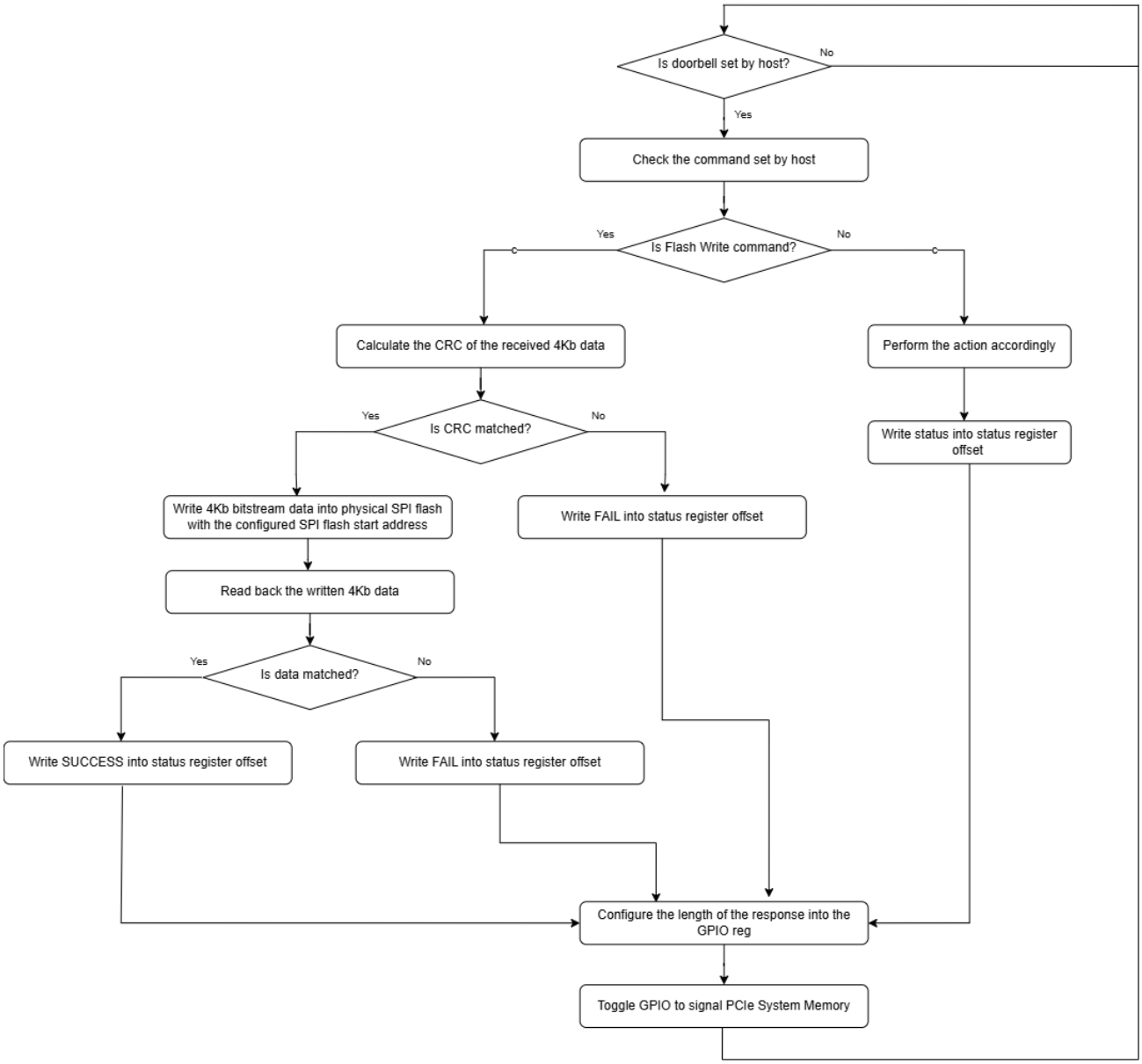


Figure 3.4. Flowchart of RISC-V Application Writing Bitstream Data to SPI Flash

The flow starts at the top with a polling decision where the RISC-V application continuously checks whether the doorbell has been set by the Host. If the doorbell is not set, the flow loops back and keeps polling. Once the doorbell is detected, the RISC-V reads and checks what command the Host has set.

If the command is not a Flash Write command, the RISC-V performs the action corresponding to whatever other command was issued, then writes the resulting status into the status register offset.

If the command is a Flash Write command, the RISC-V begins the flash update process. First, the RISC-V calculates the CRC of the received 4 kB data to verify its integrity. If the CRC does not match, the RISC-V immediately writes a FAIL status into the status register offset and skips ahead to the final steps. If the CRC does match, the RISC-V proceeds to write the 4 kB bitstream data into the physical SPI flash at the configured SPI flash start address. After writing, it reads back the written 4 kB data to verify that what was written matches what was intended. If the readback data matches, the RISC-V writes a SUCCESS status into the status register offset. If the readback data does not match, it writes a FAIL status instead.

Lastly, the RISC-V configures the length of the response into the GPIO register, then toggles the GPIO to signal the PCIe System Memory Bridge that the operation result is ready to be picked up by the Host.

3.8. FPGA Memory Write Flow

The flowchart below illustrates how the RISC-V processor reports the operation status back to the host. Once the FPGA completes its assigned operation, the PCIe System Memory Bridge transfers the status data to the host, which receives it as an interrupt at step H10 of Figure 3.2.

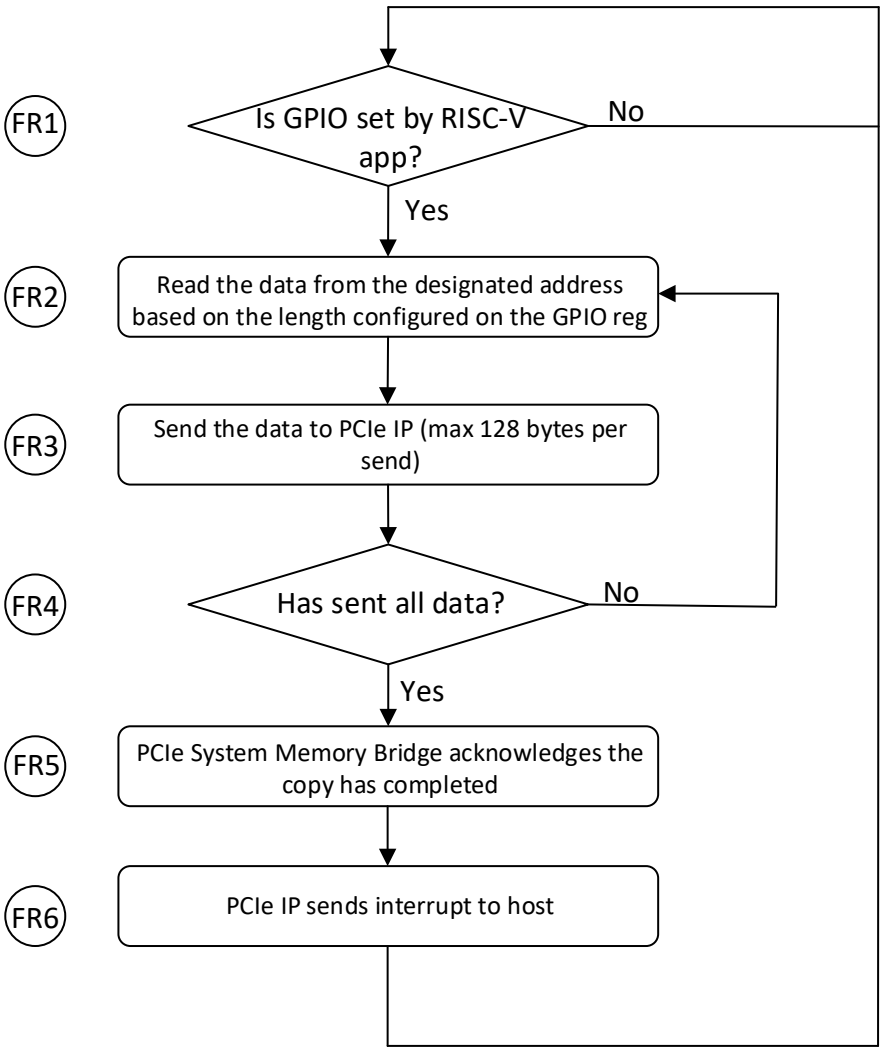


Figure 3.5. Flowchart of PCIe System Memory Bridge Data Transfer from FPGA to Host

The flow begins at *FW1* with a polling decision where the FPGA continuously checks whether the GPIO has been set by the RISC-V application. If the GPIO has not been set, the flow loops back and keeps waiting. Once the GPIO is detected as set, the flow proceeds to *FW2*, where the PCIe System Memory Bridge reads the data from the designated address in System Memory based on the transfer length that was configured in the GPIO register by the RISC-V. With the data in hand, the flow moves to *FW3*, where the PCIe System Memory Bridge sends that data to the PCIe IP in batches of up to 128 bytes per send, dispatching it toward the Host over the PCIe bus. This 128-byte limit is determined by the Maximum Payload Size (MPS) of the NXP i.MX95, which is set to 128 bytes.

After each send, the flow reaches the decision point at *FW4*, which checks whether all the data has been sent. If there is remaining data, the flow loops back to *FW2* to read and send the next batch. This loop between FW2, FW3, and FW4 continues until the entire dataset has been transmitted. Once all data has been sent, the flow proceeds to *FW5*, where the PCIe System Memory Bridge acknowledges the PCIe IP that the copy operation has completed. Finally, at *FW6*, the PCIe IP sends an interrupt to the Host to notify it that the write operation is done.

4. Running the Reference Design

This section describes the procedure for running the NXP PCIe Companion reference design using the Lattice Propel Builder software and Lattice Radiant software.

4.1. Installing the Custom IPs (Optional)

This section describes the procedure for installing custom IPs using the Lattice Propel Builder software.

To install custom IPs, perform the following steps.

1. Open the Lattice Propel Builder software, as shown in Figure 4.1.

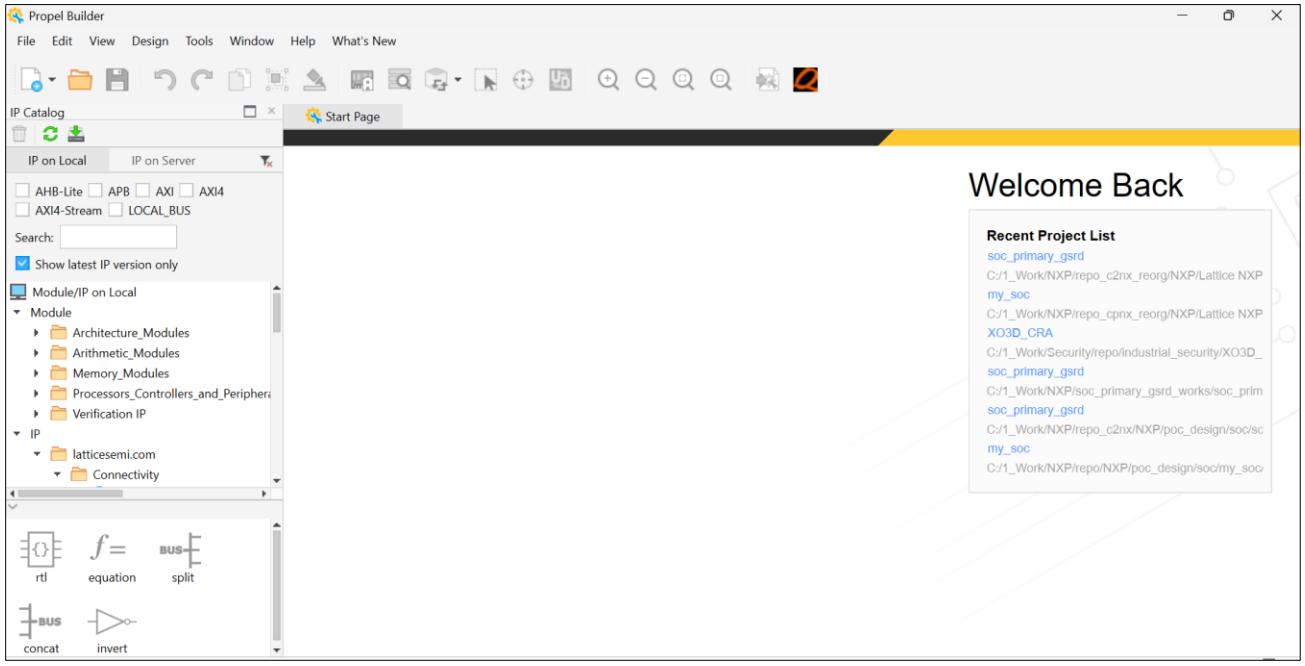


Figure 4.1. Lattice Propel Builder Software Interface

2. Right-click on any of the IP folder and select **Install IP from file**, as shown in Figure 4.2.

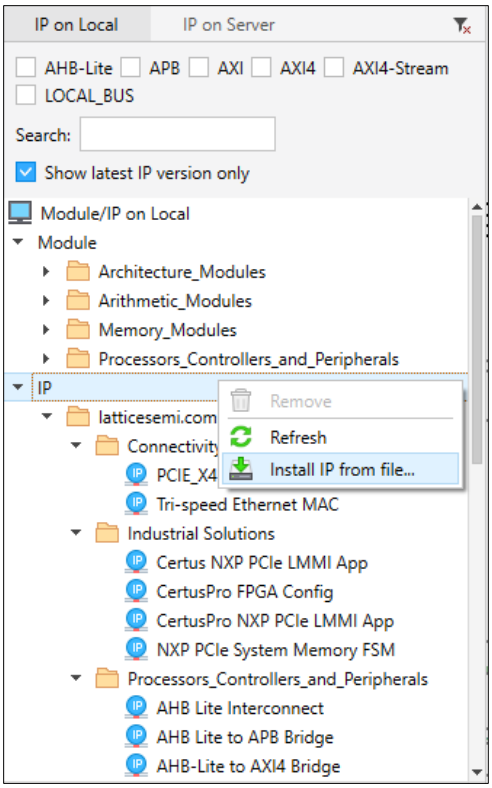


Figure 4.2. Installing a Custom IP from File

3. Select the custom IP file (.ipk) in the IPK folder. Click **Open**.

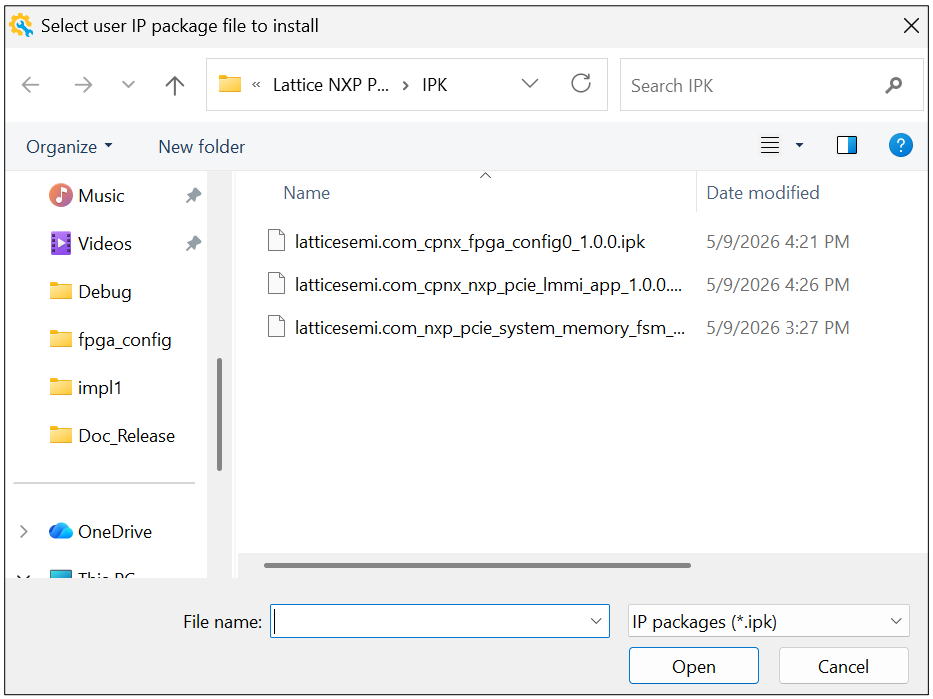


Figure 4.3. Selecting the Custom IP Package (.ipk) File

4. When the installation starts, click **Agree**.

4.2. Creating a Project in Propel Builder

This section describes the procedure for creating a reference design project using the Lattice Propel Builder software.

To create the reference design project, perform the following steps.

1. Download and install the NXP PCIe Companion Propel patch or Propel 2026.1 onwards Solution Control Pack.
2. Open the Propel Builder software.
3. Create a new SoC Design through the File menu.
4. Enter the desired project name and save location for the project.
5. Select **LFCPNX NXP PCIe Companion** for CertusPro-NX device or **LFD2NX NXP PCIe Companion** for Certus-NX device.

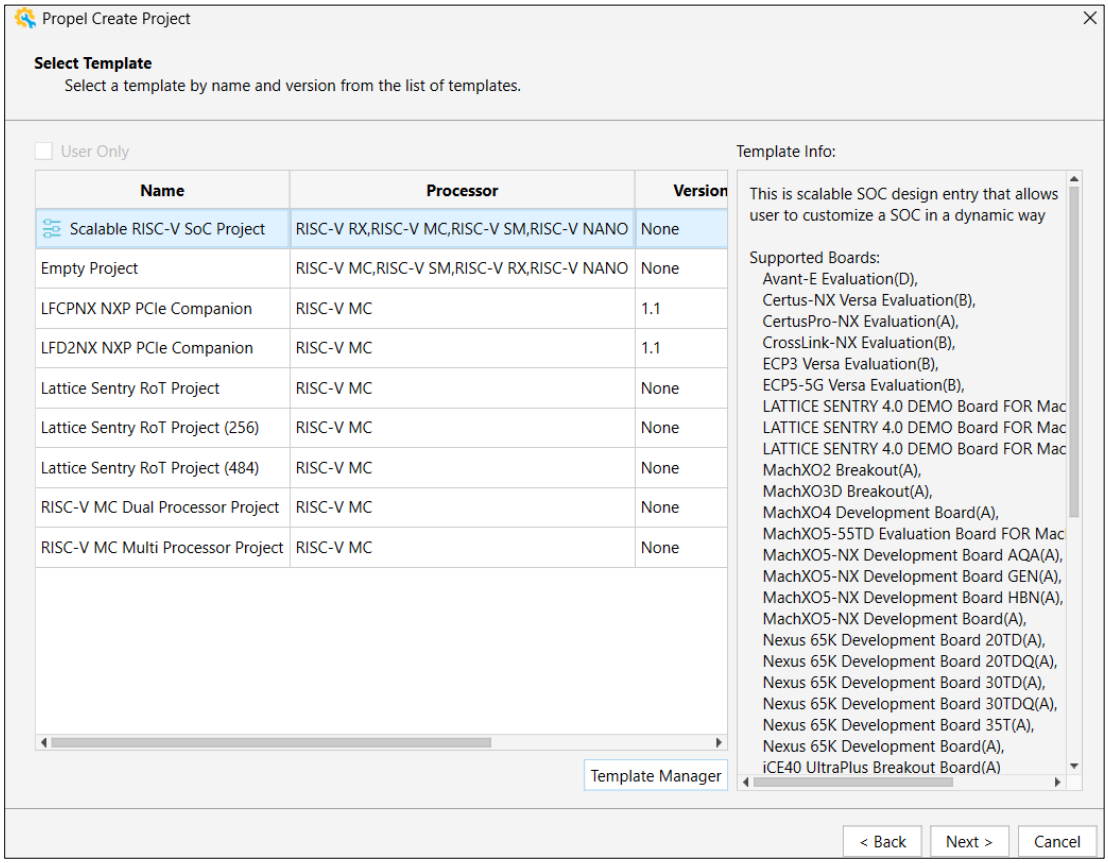


Figure 4.4. Selecting the NXP PCIe Companion Design Template

6. Click Finish at the next page to bring up the layout of the reference design.

4.3. Creating and Compiling the C Project in Propel

This section describes the procedure for creating a C project and performing compilation using the Lattice Propel Software.

To create the C project, perform the following steps.

1. In Propel Builder software, click the **Run Propel** button at the top.

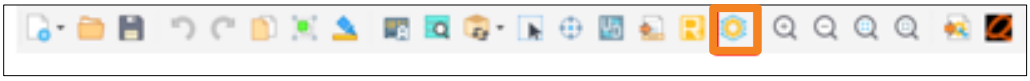


Figure 4.5. Launching the Propel Software

2. Wait for Propel to launch and the C/C++ Project window to appear.
3. In the Example Application section, choose the option that corresponds to your device. For the CertusPro-NX PCIe Bridge Board, select **CPNX NXP PCIe Companion Application**, while for the Certus-NX Board, select **CNX NXP PCIe Companion Application**.

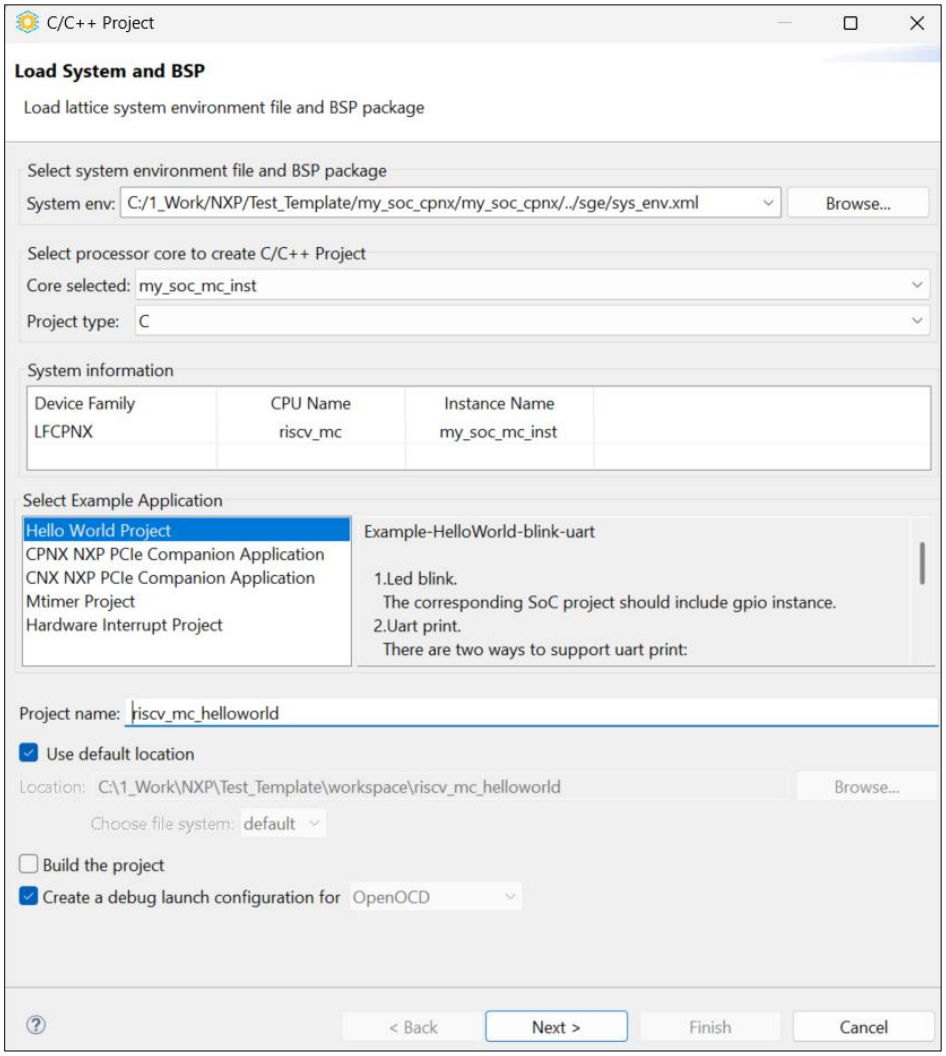


Figure 4.6. Creating a New C Project in Propel

4. Enter the desired project name and click **Next**.
5. (Optional) To enable UART log printing, in Lattice Toolchain Setting, select the C/C++ Compiler tab and enable the Enable UART function (-DLSCC_STDIO_UART_APB) option.
6. Click **Next**.
7. To compile the project, click the Builder button, or right-click on the project and select the Build Project.
   - For better performance, compile the project with O1 optimization by right clicking on the project, navigating to **Properties > C/C++ Build > Settings > Optimization**, and selecting Optimize (-O1) for Optimization Level.

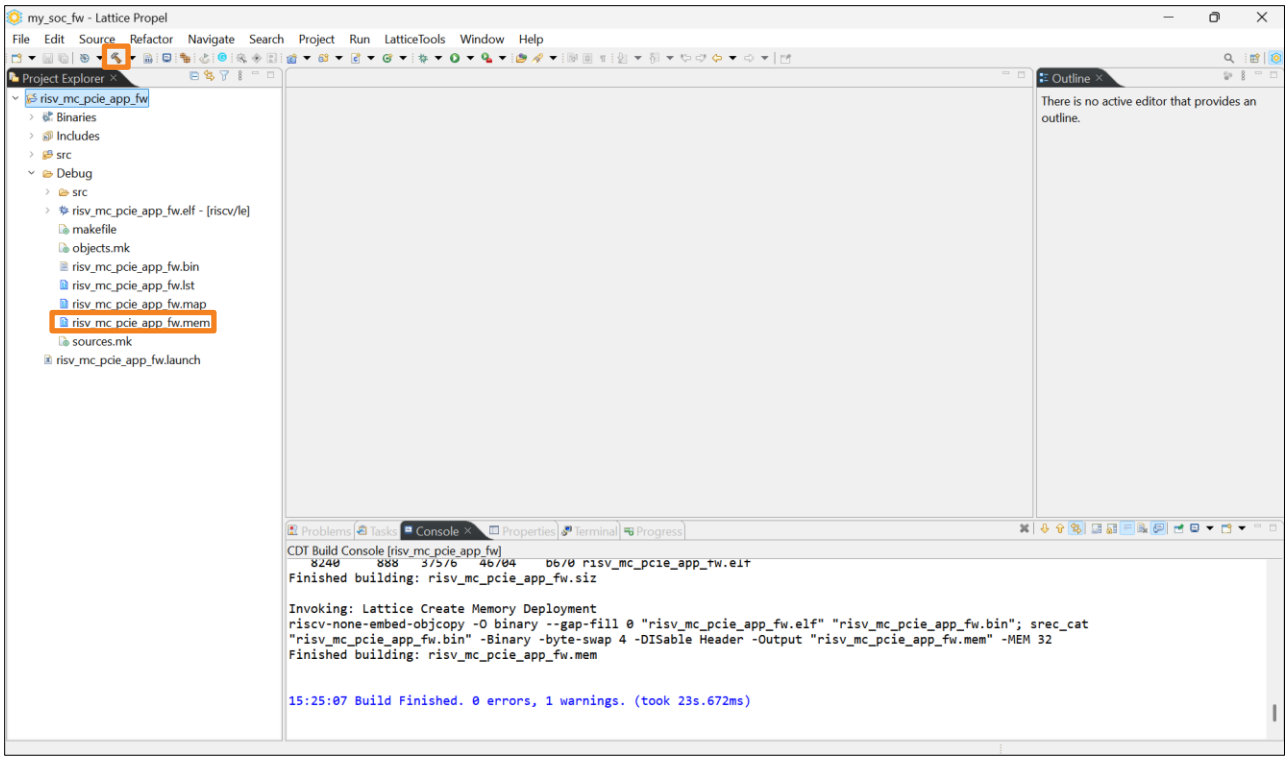


Figure 4.7. Compiling the C Project.

8. Wait for the compilation to complete. The .mem file is generated in the Debug folder.

4.4. Generating the Bitstream File

This section describes the procedure for generating the FPGA bitstream file using the Lattice Radiant Software.

To generate the FPGA bitstream file, perform the following steps.

1. Go back to Propel Builder.
2. Right-click on the System Memory 0 component IP and select **Reconfig**.
3. Set the Memory Initialization property value to **Memory File**.

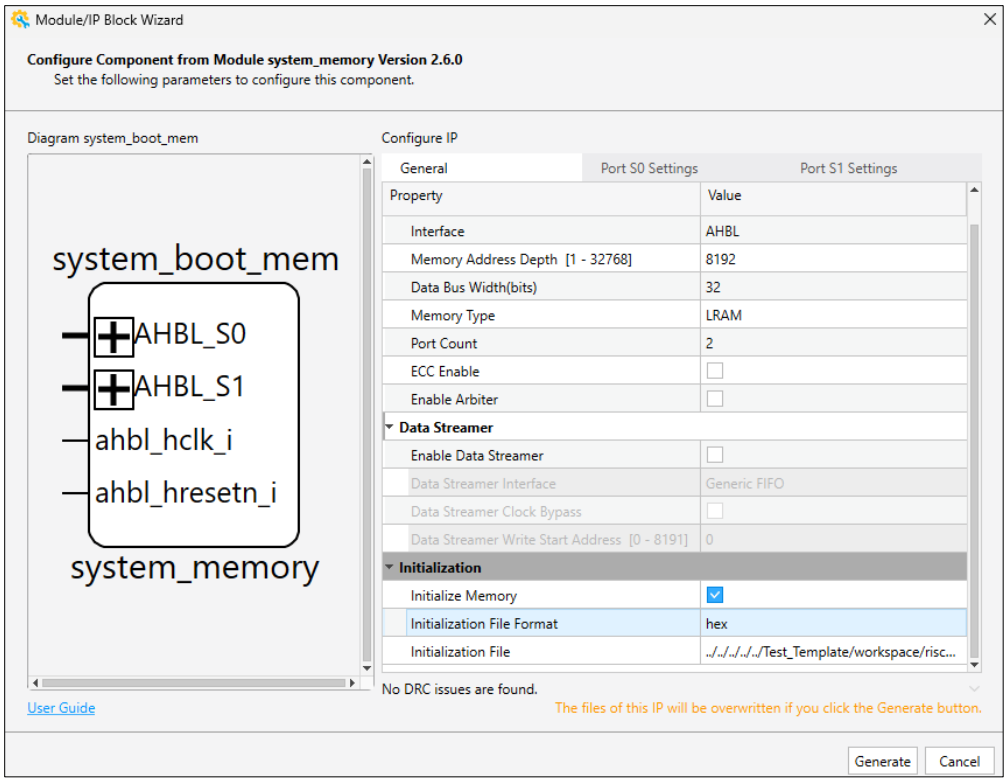


Figure 4.8. Updating System Memory 0 with the Generated .mem File

4. For the Memory File property, select the .mem file previously generated by Propel.
5. Click the Generate button at the bottom right.
6. Save the design and click the **Run Radiant** button located in the top menu bar. Wait for the design to be loaded in Radiant.

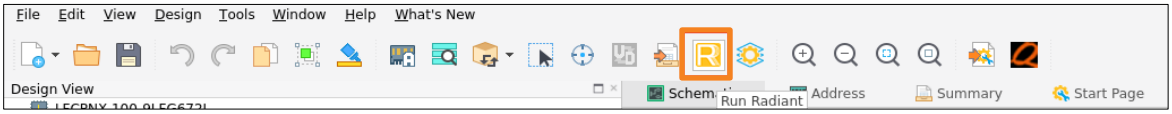


Figure 4.9. Lattice Radiant Software Interface

7. Click **Export Files** to generate the bit file. Review the log message in the Export Reports folder for the generated bitstream.

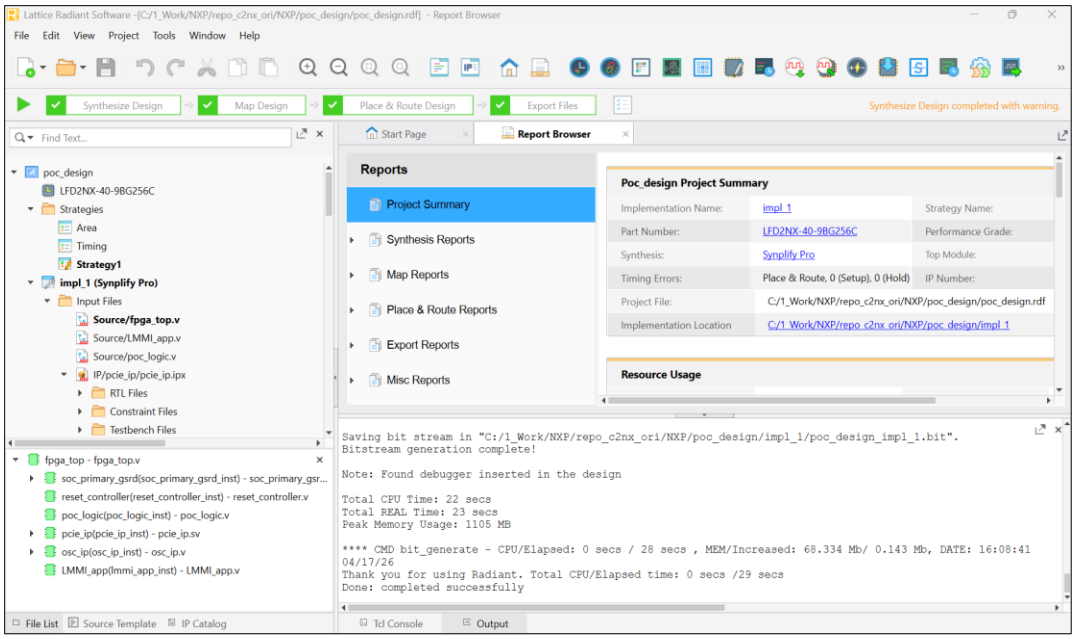


Figure 4.10. Report Log Confirming Successful Bitstream Generation

4.5. Generating the MCS File

Generating an MCS file enables the FPGA to fall back to the golden image automatically if the primary bitstream becomes corrupted during a host-initiated update.

To generate an MCS file, perform the following steps.

1. In Lattice Radiant Programmer, navigate to **Tools > Deployment Tool**.
2. Set the Function Type to **External Memory** and the Output File Type to **Advanced SPI Flash**, then click **OK**.

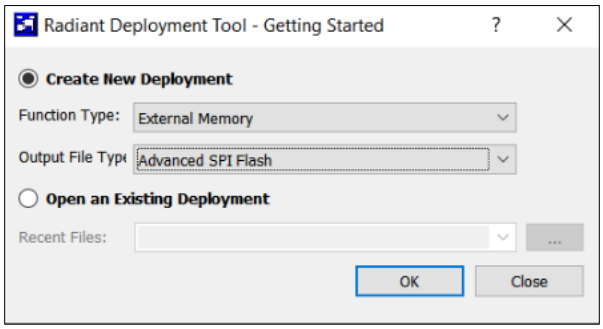


Figure 4.11. Creating a New Deployment in the Radiant Deployment Tool

3. Click the file name field to browse and select the bitstream file that serves as the primary bitstream, then click **Next**.

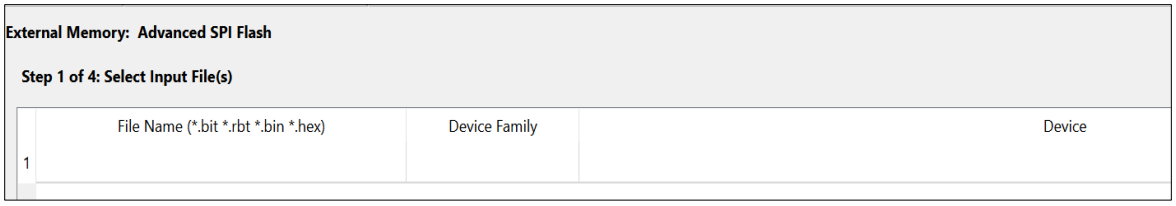


Figure 4.12. Selecting the Primary Bitstream File

4. Navigate to the **Multiple Boot** tab.
5. Enable the **Multi-Boot** option.
6. Click the **Golden Pattern** browse button to select the bitstream file that serves as the golden image.
7. The starting address of the Golden Pattern is assigned automatically. To use a different address, select an alternative value from the drop-down menu.

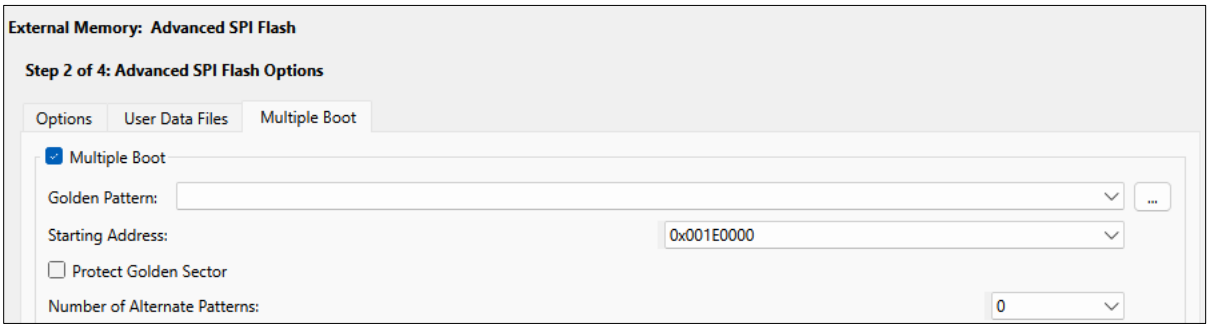


Figure 4.13. Selecting the Golden Image Bitstream for Multi-Boot Fallback

8. Select **Next**.
9. Enter the desired name for the output PROM hex file in the Output File 1 field, then click **Next**.
10. Review the configuration summary, then click Generate to produce the MCS file.

4.6. Programming the Bitstream File

The bitstream is deployed to the FPGA using the Programmer utility. The figures below provide the specific device details as they appear within the application.

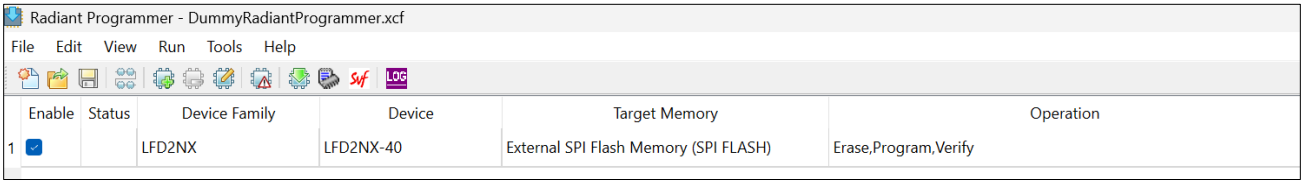


Figure 4.14. Certus-NX Device Configuration in Lattice Programmer

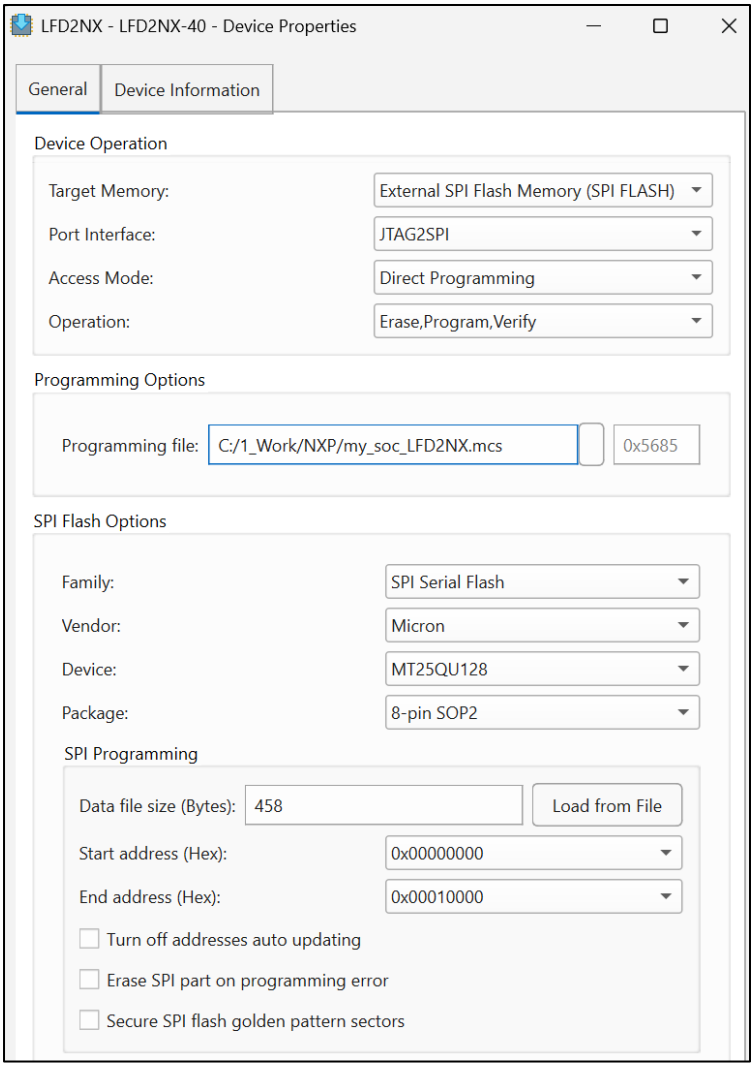


Figure 4.15. Certus-NX SPI Flash Device Settings in Lattice Programmer

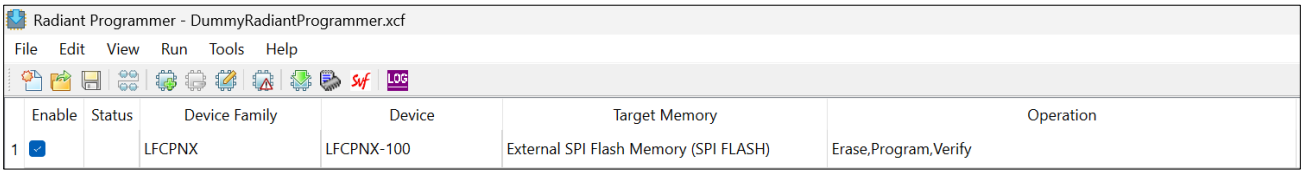


Figure 4.16. CertusPro-NX PCIe Bridge Board Device Configuration in Lattice Programmer

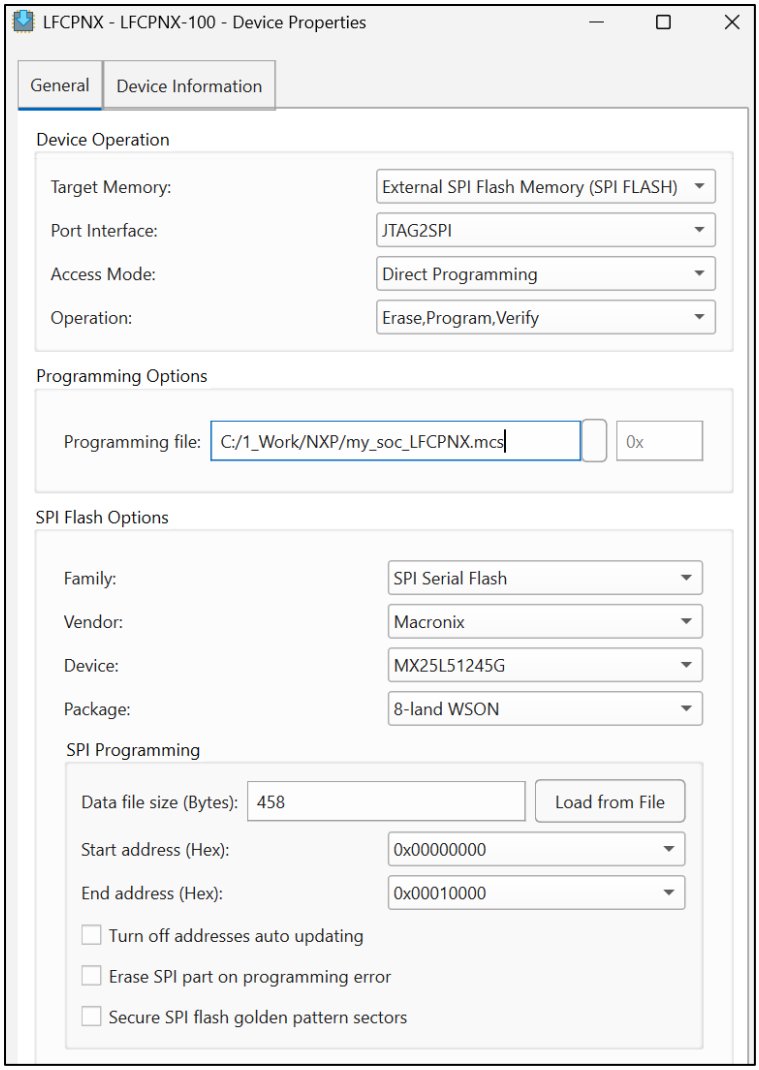


Figure 4.17. CertusPro-NX PCIe Bridge Board SPI Flash Device Settings in Lattice Programmer

5. Implementing the Reference Design on Board

Figure 5.1 shows the NXP i.MX95 evaluation kit board.

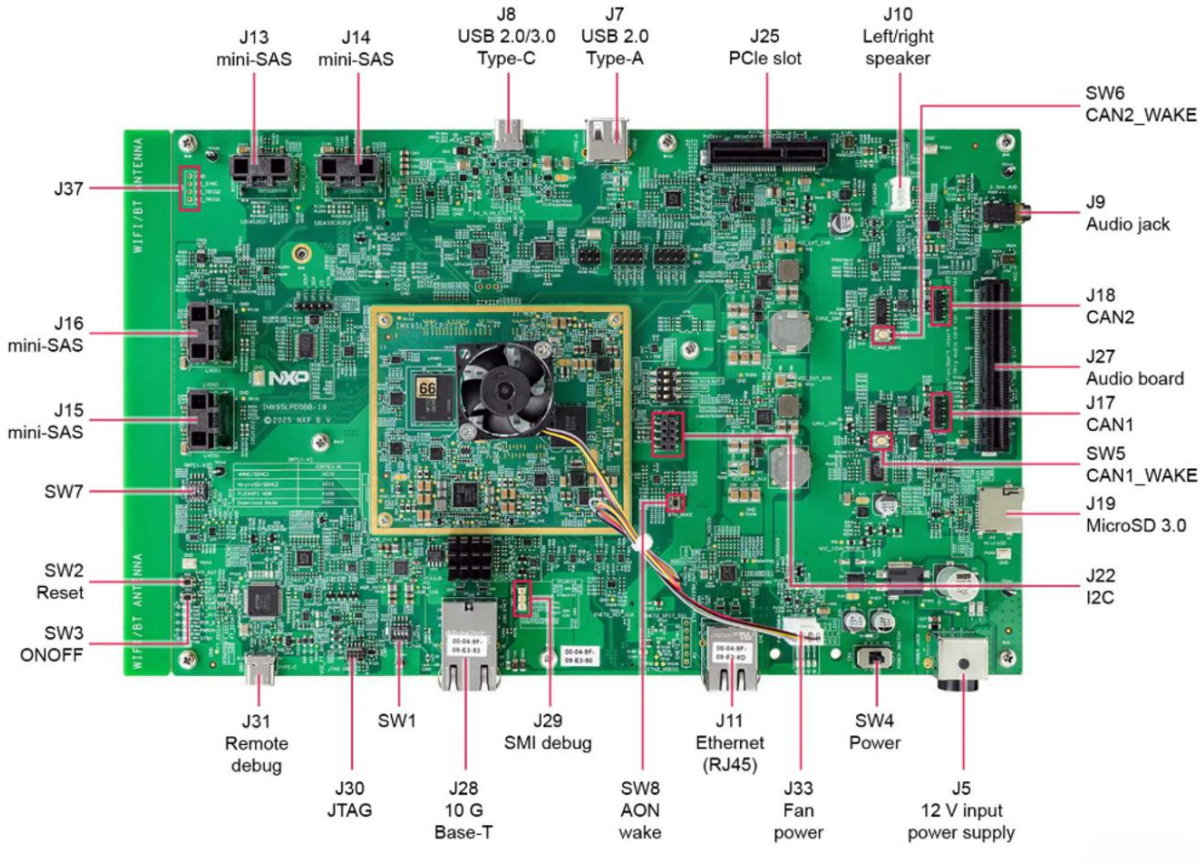


Figure 5.1. NXP i.MX95 Evaluation Kit Board Overview

5.1. Setting Up the NXP i.MX95 Board

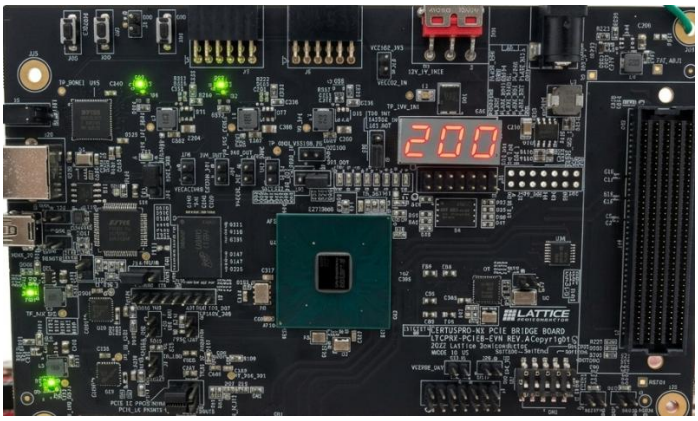
To set up the NXP i.MX95 board, perform the following:

1. Verify the jumper configurations on the device according to the table below.

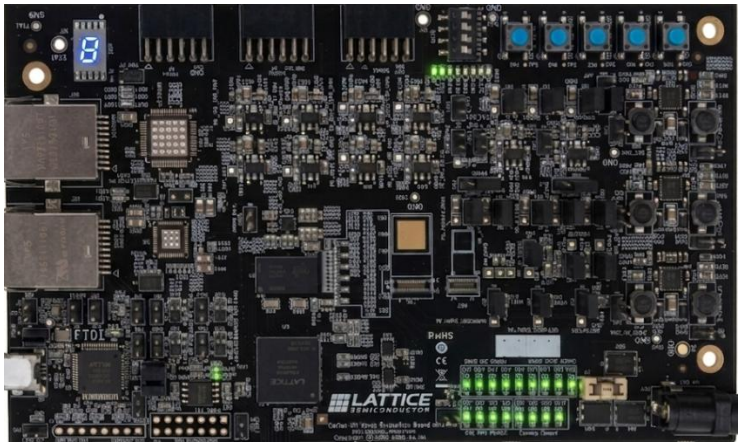
Table 5.1. FPGA Jumper Configuration

Device	Jumper Configuration
Certus-NX	JP23 – Short (Default)
CertusPro-NX PCIe Bridge board	J4 – Short 2-4 (Default)

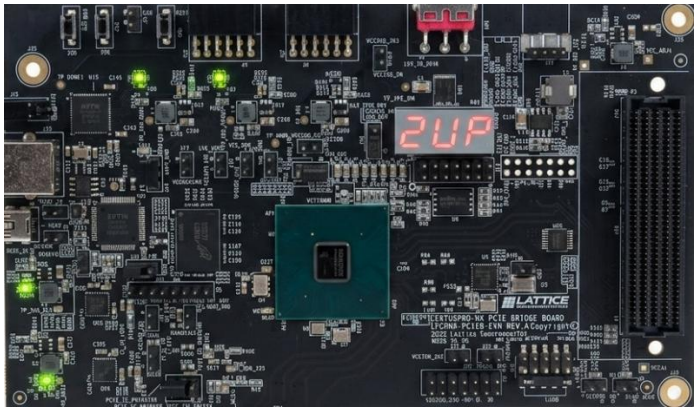
2. Insert the FPGA card into the PCIe slot (J25).
3. Connect a USB cable from PC to the UART debug port for kernel log monitoring.
   - Upon successful connection, the four COM ports appears in your system. If no COM ports are detected, disconnect the USB cable, then reconnect it. If the issue persists, replace the USB cable with a new one.
   - Open all detected COM ports using a terminal application such as PuTTY or Tera Term.
4. Connect the i.MX95 power supply and switch it ON.
5. Turn on the i.MX95 power switch.
6. When the FPGA PCIe link is down,
   - For CertusPro-NX PCIe Bridge Board, “00” is shown on the 7-segment display.



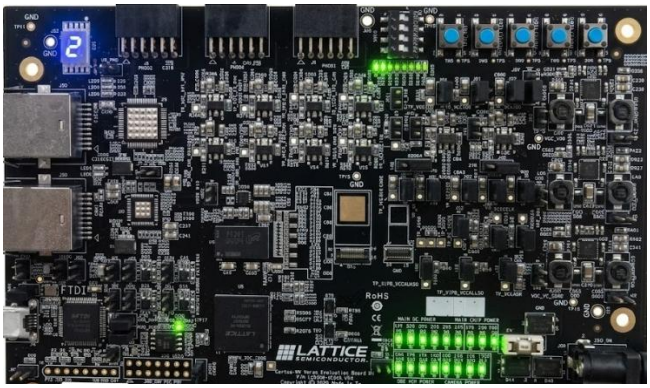
- For Certus-NX, the value *8* is shown on the 7-segment display.



7. When the FPGA PCIe link is up,
   - For CertusPro-NX PCIe Bridge Board, *UP* is shown on the 7-segment display.



   - For Certus-NX, the value *1* or *2* is shown on the 7-segment display.



- When using the provided reference design MCS file, a '3' on the 7-segment display signifies a primary bitstream corruption. In this state, the FPGA device automatically falls back to boot from the golden bitstream.
- Note that the power switch on the CertusPro-NX PCIe Bridge Board must be in off position for the board to be detected by the host system.

8. The PCIe link configuration used throughout this demo is GEN2 running at 5 GT/s with a single x1 lane, Bridge mode, single BAR. The Maximum Payload Size is set to 128 bytes and the Maximum Read Request Size is 512 bytes, with a PHY reference clock of 100 MHz.
9. For the Certus-NX board, communication goes through the Lattice PCIe x1 IP Core, while the CertusPro-NX PCIe Bridge Board uses the Lattice PCIe x4 IP Core configured at x1 lane width. Both devices operate in Bridge Mode with an AXI-MM interface.
10. Connect the USB cable to Remote debug port (J31).
11. Launch a terminal application and establish a connection using a baud rate of 115200.
12. During boot, one of the terminals displays the kernel log output.
13. In the terminal when prompted for login information, log in using the following credentials:
    - Username: root
    - Password: (leave blank)
14. Copy the following files to the root directory of the i.MX95 using a USB storage device:

Table 5.2. Files to Copy to i.MX95

File	Description
c2nx_bitstream_1.bit cpnx_bitstream_1.bit	Bitstream to be programmed to the FPGA per user selection. The value 1 is shown on the 7-segment display.
c2nx_bitstream_2.bit cpnx_bitstream_2.bit	Bitstream to be programmed to the FPGA per user selection. The value 2 is shown on the 7-segment display.
lattice_nxp_pcie_ep.ko	Lattice FPGA PCIe endpoint driver. The driver is compiled on version 6.12.49 (df24f9428e38) of the NXP Linux.
c2nx_bwop.o cpnx_bwop.o	Application that handles the bitstream update flow on i.MX95.

5.2. Bitstream Update User Interface Application

To update the bitstream user interface, perform the following:

1. Launch the user interface application.
   a. For CertusPro-NX PCIe Bridge board, launch LFCPNX_PcieFpgaUpdateGUI.exe.
   b. For Certus-NX, launch LFD2NX_PcieFpgaUpdateGUI.exe.
2. When prompted, enter the COM port number that corresponds to the terminal displaying the kernel log output.
   - Ensure that no other application is using the selected UART COM port. Otherwise, the user interface fails to connect.

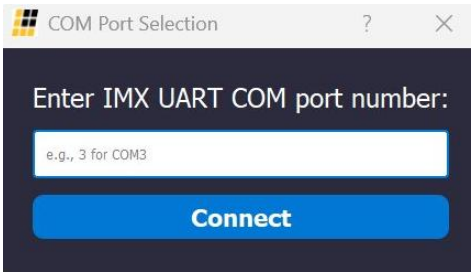


Figure 5.2. COM Port Selection in the Bitstream Update User Interface Application

3. Click **Connect**. The user interface attempts to log in to the i.MX95 automatically.
4. Once connected, the FPGA Update Selection screen is displayed.

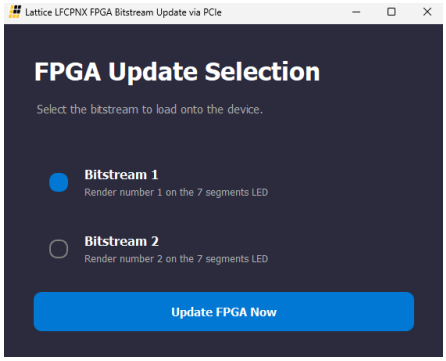


Figure 5.3. FPGA Update Selection Screen of the User Interface Application

5. Select the desired bitstream and click **Update FPGA Now**.
6. The user interface displays the update progress during bitstream transfer.

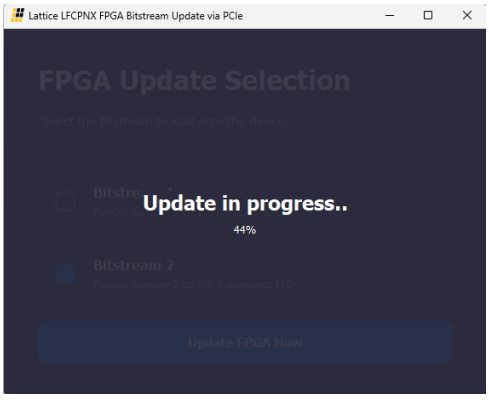


Figure 5.4. Bitstream Transfer Progress Displayed in the User Interface Application

7. Upon successful completion of the update, the system triggers FPGA reconfiguration automatically to apply the new bitstream.

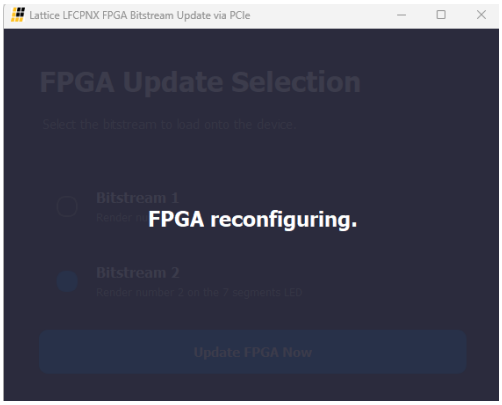


Figure 5.5. Automatic FPGA Reconfiguration Upon Successful Bitstream Update

8. Following FPGA reconfiguration, the selected value is displayed on the FPGA 7-segment display.

6. Latency Benchmarking

The design incorporates a hardware-based profiling mechanism for measuring PCIe latency. By monitoring specific General Purpose Input/Output (GPIO) signals on both the i.MX95 and the FPGA, the precise data transfer timings can be determined.

Table 6.1. Upstream to Downstream Copy Latency

Device	Signal Name	Ball Location
CertusPro-NX PCIe Bridge Board	J8 PMOD1_1	T1
Certus-NX	J8 PMOD2_1	H1

The preceding pins are asserted when the FPGA is copying data from host to FPGA system memory. This pin stays asserted until the copy operation is complete.

Table 6.2. Downstream to Upstream Copy Latency

Device	Signal Name	Ball Location
CertusPro-NX PCIe Bridge Board	J8 PMOD1_2	T2
Certus-NX	J8 PMOD2_2	H2

The preceding pins are asserted when the FPGA is transferring data from FPGA system memory to the host. This pin stays asserted until the copy operation is complete.

Table 6.3. PCIe System Memory Bridge Module AXI Write

Device	Signal Name	Ball Location
CertusPro-NX PCIe Bridge Board	J8 PMOD1_3	U3
Certus-NX	J8 PMOD2_3	N1

The preceding pins are toggled whenever the PCIe System Memory module receives a write request from PCIe IP. Once asserted, the pins remain high for 40 clock cycles before deasserting.

Table 6.4. RISC-V

Device	Signal Name	Ball Location
CertusPro-NX PCIe Bridge Board	J8 PMOD1_4	U6
Certus-NX	J8 PMOD2_4	K5

These pins are asserted when the RISC-V processor receives a trigger from the host to begin processing based on the command issued by the host. The assertion is maintained until the RISC-V processor has completed the operation.

i.MX95

Latency benchmarking can be performed by running the gpio_latency.c application on the i.MX95. To use this application, modification is required on the i.MX95 Linux kernel. Specifically, the I2C6 SDA and I2C6 SCL lines (GPIO_IO02 and GPIO_IO03 in schematic reference, or gpiochip3 pin 2 and gpiochip3 pin 3 in Linux) are repurposed as general-purpose I/O pins for the toggling operation. This requires updating the i.MX95 Device Tree Blob (DTB) to reassign these pins from their default function to GPIO mode.

- **GPIO_IO02** – Asserted when the host triggers the doorbell to notify the FPGA to begin a data transfer operation, either copying data from the host to the FPGA system memory or sending data to host from the FPGA. The signal remains asserted until the operation is complete.
- **GPIO_IO03** – Serves the same purpose as GPIO_IO2, with the addition of the time taken to configure the necessary registers for the operation prior to triggering it.

The latency measurements presented in this section were captured under the following PCIe link configuration: GEN2 (5 GT/s) link speed, x1 lane width, single BAR, a Maximum Payload Size (MPS) of 128 bytes, and a Maximum Read Request Size (MRRS) of 512 bytes. The PHY reference clock frequency is 100 MHz. All measurements were performed on the NXP i.MX95 host running a Yocto-based Linux environment. Certus-NX communicates with the host through the Lattice PCIe x1 IP Core, while CertusPro-NX communicates through the Lattice PCIe x4 IP Core configured with a x1 lane width. Both devices operate in Bridge Mode with an AXI-MM interface.

6.1. FPGA-to-Host Latency Benchmarking

The flowchart below illustrates the sequence of FPGA-to-Host memory read and write operations as captured during latency benchmarking.

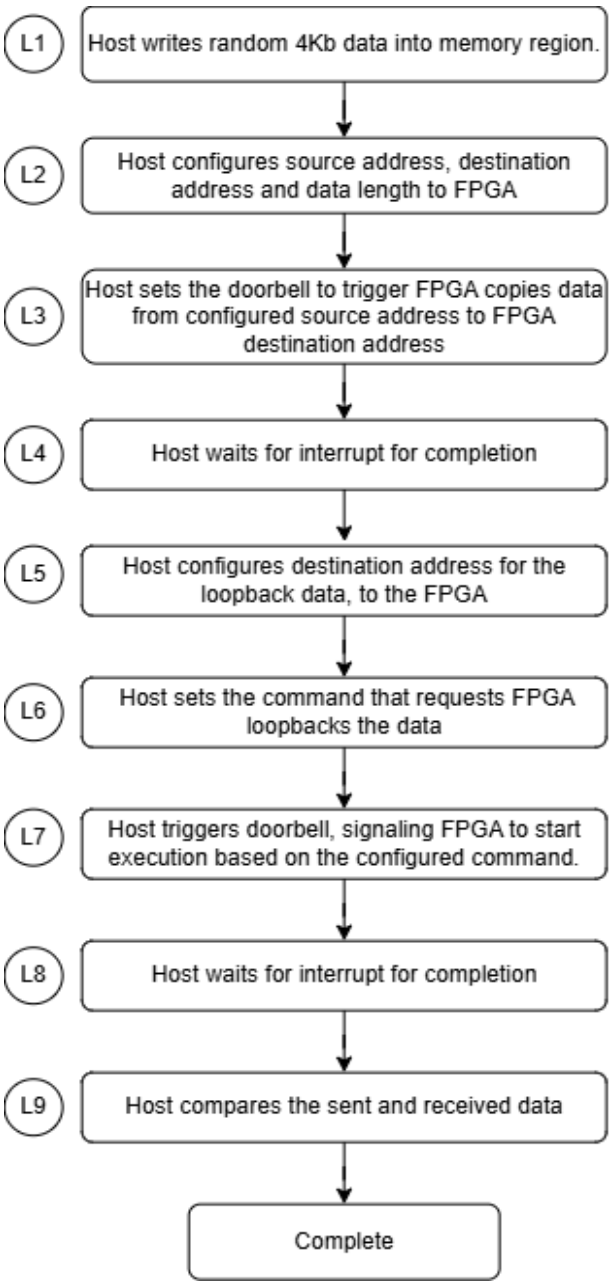


Figure 6.1. Flowchart of FPGA-to-Host Memory Read and Write Operations for Latency Benchmarking

Figure 6.2 shows the measured latency when the FPGA performs a 4 kB Memory Read from the host.

Labels L1–L4 correspond directly to the matching steps in Figure 6.1 while *Labels FR1* and *FR2* correspond to the matching steps in Figure 3.3.

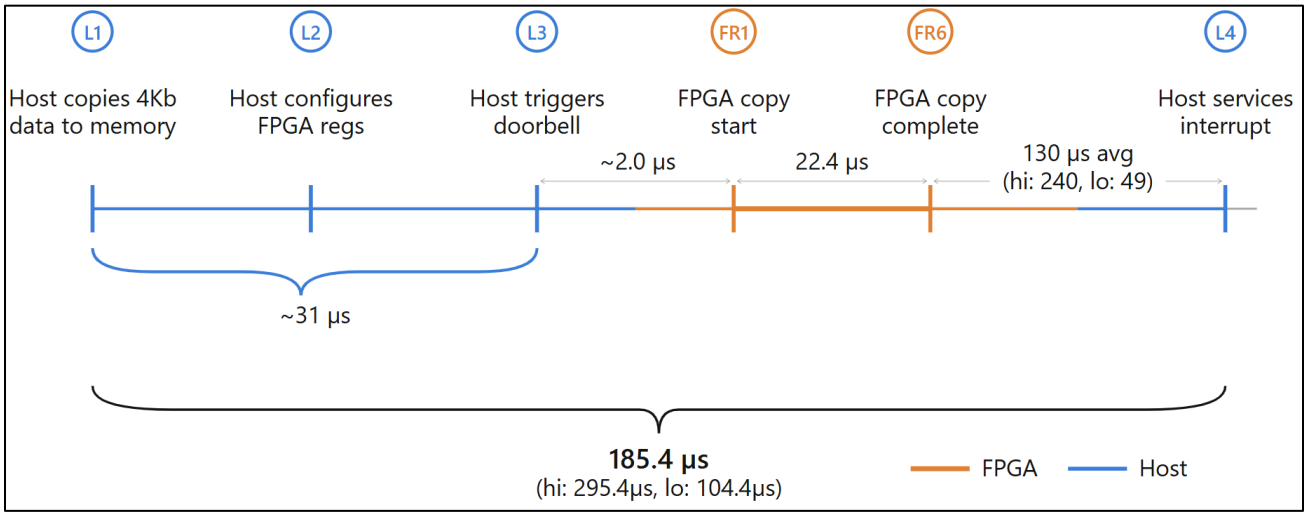


Figure 6.2. Measured Latency of FPGA Copying 4 kB of Data from Host to System Memory

Figure 6.3 shows the measured latency when the FPGA performs a 4 kB Memory Write to the host.

Labels L5–L8 correspond directly to the matching steps in Figure 6.1 while *Labels FW1* and *FW6* correspond to the matching steps in Figure 3.5.

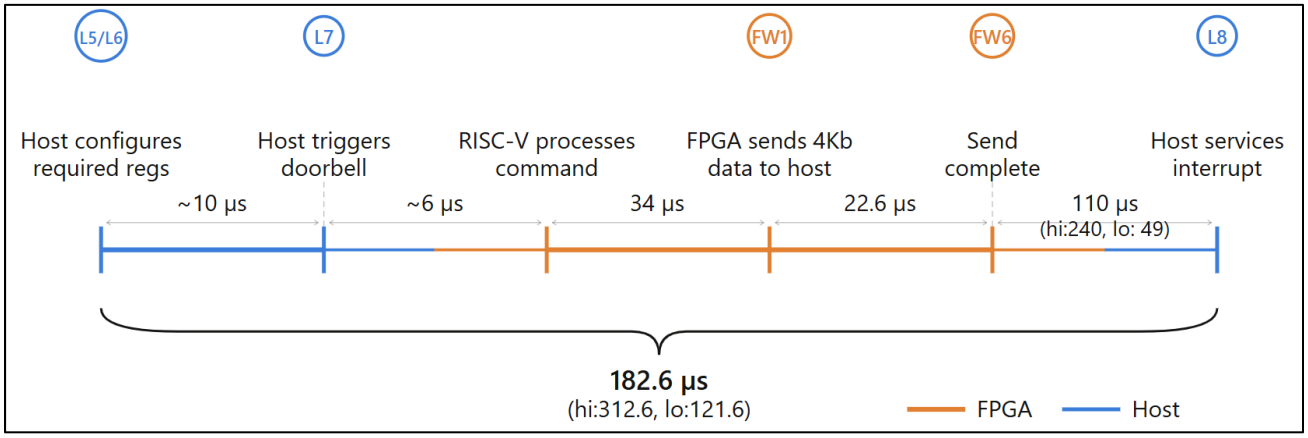


Figure 6.3. Measured Latency of FPGA Writing 4 kB of Data from System Memory to Host

6.2. Host-to-FPGA Latency Benchmarking

The flowchart below illustrates the sequence of Host-to-FPGA memory read and write operations as captured during latency benchmarking.

Note that the first PCIe write transaction after a period of host inactivity may take approximately 5–6 µs to complete. This elevated latency is a one-time overhead that occurs only on the initial write, after which subsequent write transactions complete significantly faster. This behavior is observed consistently on both devices. The latency figures presented below reflect the steady-state subsequent write latency representative of normal operating conditions.

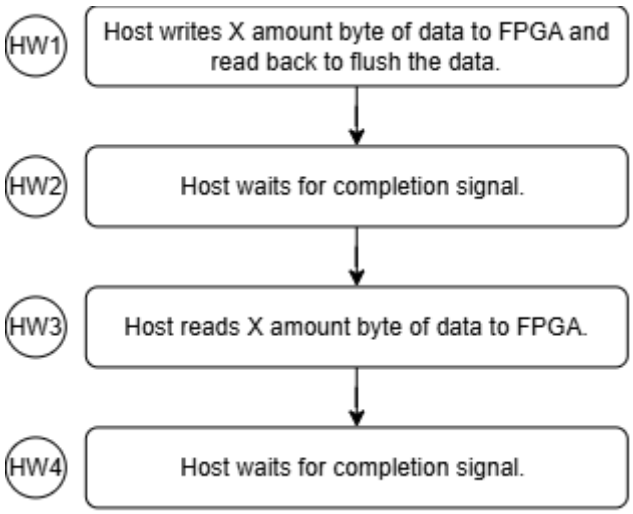


Figure 6.4. Flowchart of Host-to-FPGA Memory Read and Write Operations for Latency Benchmarking

Figure 6.5 shows the measured latency when the host performs a 4-byte memory write to the FPGA and read back. **Labels HW1–HW4** correspond directly to the matching steps in Figure 6.4.

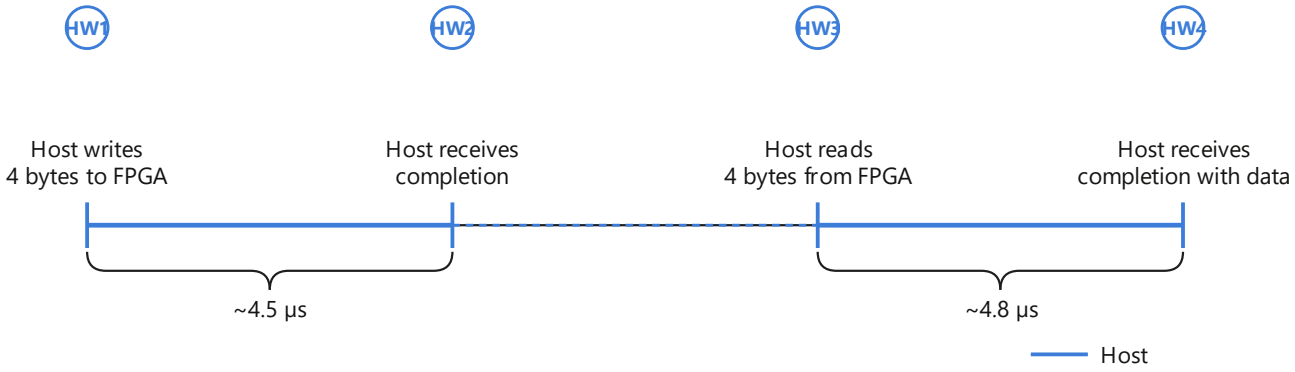


Figure 6.5. Measured Latency of Host-to-FPGA Memory Write and Read-Back for 4-Byte Transactions

Figure 6.6 shows the measured latency when the host performs an 8-byte memory write to the FPGA and read back. *Labels HW1–HW4* correspond directly to the matching steps in Figure 6.4.

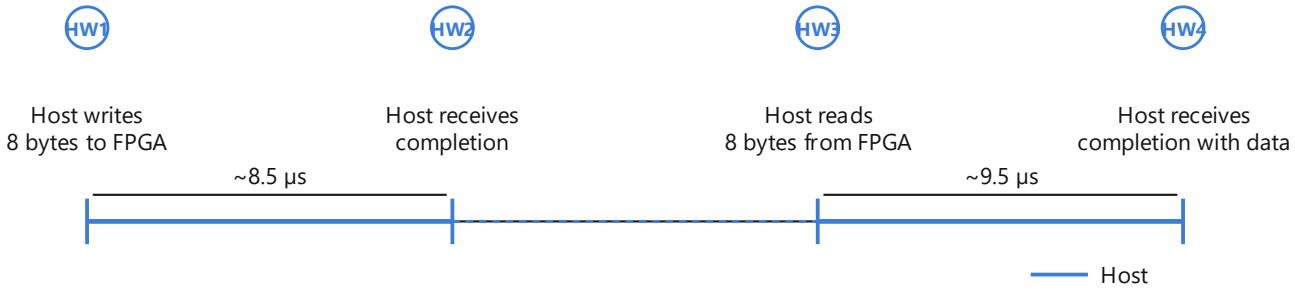


Figure 6.6. Measured Latency of Host-to-FPGA Memory Write and Read-Back for 8-Byte Transactions

7. Resource Utilization

Map Resource Usage

impl_1	LUT4	Logic	Distributed RAM	Ripple Logic	PFU Registers	IO Buffers	EBR
my_soc	25183(5)	17493(5)	4836(0)	2854(0)	15873(2)	29(21)	63(0)
usr_lmmi_reset_inst	0(0)	0(0)	0(0)	0(0)	1(1)	0(0)	0(0)
uart0_inst	623(0)	535(0)	12(0)	76(0)	457(0)	0(0)	0(0)
sysmem1_inst	1385(0)	923(0)	222(0)	240(0)	746(0)	0(0)	16(0)
sysmem0_inst	702(0)	458(0)	198(0)	46(0)	250(0)	0(0)	32(0)
spi_controller0_inst	2403(0)	2221(0)	6(0)	176(0)	1706(0)	4(0)	2(0)
seven_seg_led_control_inst	40(40)	18(18)	12(12)	10(10)	91(91)	0(0)	0(0)
reset_controller_inst	0(0)	0(0)	0(0)	0(0)	2(2)	0(0)	0(0)
pll_clk_inst	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
pcie_sys_mem_bridge_inst	1197(0)	643(0)	0(0)	554(0)	727(0)	0(0)	0(0)
pcie_ip_inst	5531(1)	3775(1)	1062(0)	694(0)	4084(0)	0(0)	11(0)
osc_wrapper_inst	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
mc_inst	3099(0)	2665(0)	0(0)	434(0)	1526(0)	0(0)	2(0)
lmmi_app_inst	46(0)	38(0)	0(0)	8(0)	23(0)	0(0)	0(0)
jtaghub_top_inst	34(33)	20(19)	0(0)	14(14)	53(53)	4(4)	0(0)
gpio0_inst	303(0)	303(0)	0(0)	0(0)	265(0)	0(0)	0(0)
gen_pcie_op_compl_int_inst	21(21)	11(11)	0(0)	10(10)	12(12)	0(0)	0(0)
fpga_config_inst	55(0)	55(0)	0(0)	0(0)	55(0)	0(0)	0(0)
debounce_inst	78(78)	44(44)	0(0)	34(34)	36(36)	0(0)	0(0)
axi2apb1_inst	244(0)	202(0)	0(0)	42(0)	192(0)	0(0)	0(0)
axi2ahb_0_inst	618(0)	536(0)	54(0)	28(0)	212(0)	0(0)	0(0)
axi_interconnect0_inst	8653(0)	4895(0)	3270(0)	488(0)	5361(0)	0(0)	0(0)
apb_interconnect0_inst	61(0)	61(0)	0(0)	0(0)	4(0)	0(0)	0(0)
ahb2axi_0_inst	85(0)	85(0)	0(0)	0(0)	68(0)	0(0)	0(0)

Figure 7.1. Resource Utilization Summary for CertusPro-NX PCIe Bridge Board

Map Resource Usage

impl_1	LUT4	Logic	Distributed RAM	Ripple Logic	PFU Registers	IO Buffers	DSP MULT	EBR	Large RAM
my_soc	22068(2)	14820(2)	4296(0)	2952(0)	14016(0)	24(16)	6(0)	15(0)	2(0)
usr_lmmi_reset_inst	0(0)	0(0)	0(0)	0(0)	1(1)	0(0)	0(0)	0(0)	0(0)
system_uart_inst	246(0)	200(0)	0(0)	46(0)	147(0)	0(0)	0(0)	0(0)	0(0)
system_ospi_fc_inst	1513(0)	1169(0)	102(0)	242(0)	1243(0)	4(0)	0(0)	2(0)	0(0)
system_gpio_inst	270(0)	270(0)	0(0)	0(0)	265(0)	0(0)	0(0)	0(0)	0(0)
sys_mem_1_inst	1402(0)	940(0)	222(0)	240(0)	746(0)	0(0)	0(0)	0(0)	1(0)
sys_mem_0_inst	695(0)	451(0)	198(0)	46(0)	318(0)	0(0)	0(0)	0(0)	1(0)
seven_seg_led_control_inst	1(1)	1(1)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
riscv_cpu_inst	3730(0)	3014(0)	0(0)	716(0)	1919(0)	0(0)	6(0)	2(0)	0(0)
reset_controller_inst	11(11)	3(3)	0(0)	8(8)	7(7)	0(0)	0(0)	0(0)	0(0)
pcie_sys_mem_bridge_inst	1172(0)	618(0)	0(0)	554(0)	725(0)	0(0)	0(0)	0(0)	0(0)
pcie_ip_inst	5481(1)	3719(1)	1062(0)	700(0)	3985(0)	0(0)	0(0)	11(0)	0(0)
osc_wrapper_inst	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
lmmi_app_inst	33(0)	33(0)	0(0)	0(0)	20(0)	0(0)	0(0)	0(0)	0(0)
jtaghub_top_inst	34(33)	20(19)	0(0)	14(14)	53(53)	4(4)	0(0)	0(0)	0(0)
gen_pcie_op_compl_int_inst	21(21)	11(11)	0(0)	10(10)	12(12)	0(0)	0(0)	0(0)	0(0)
fpga_config_inst	56(0)	56(0)	0(0)	0(0)	55(0)	0(0)	0(0)	0(0)	0(0)
axi_to_apb_inst	245(0)	203(0)	0(0)	42(0)	190(0)	0(0)	0(0)	0(0)	0(0)
axi_to_ahb_0_inst	590(0)	508(0)	54(0)	28(0)	210(0)	0(0)	0(0)	0(0)	0(0)
axi_interconnect_0_inst	6425(0)	3461(0)	2658(0)	306(0)	4048(0)	0(0)	0(0)	0(0)	0(0)
apb_ic_inst	58(0)	58(0)	0(0)	0(0)	4(0)	0(0)	0(0)	0(0)	0(0)
ahbl_to_axi4_bridge_inst	83(0)	83(0)	0(0)	0(0)	68(0)	0(0)	0(0)	0(0)	0(0)

Figure 7.2. Resource Utilization Summary for Certus-NX

8. Upgrading and Customizing the Reference Design

8.1. Expanding I/O Expander

To integrate an additional peripheral controller into the design and enable the RISC-V application to communicate with it, follow the steps below.

8.1.1. Update the Propel Builder Design

In Propel Builder, add the peripheral controller IP to the design by dragging and dropping it onto the canvas. To enable the RISC-V application to access the peripheral controller, connect the controller to the RISC-V processor through the existing interface bridge.

If the IP controller uses an APB interface, it can be connected to the existing APB interconnect block available in the design, as shown below.

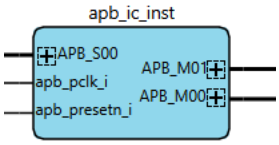


Figure 8.1. APB Interconnect Instance

If the IP controller uses an AHB interface, a new AHBL interconnect must be created and inserted between the existing AXI-to-AHB bridge and System Memory 0. Both the new peripheral controller and System Memory 0 should then be connected to this AHBL interconnect.

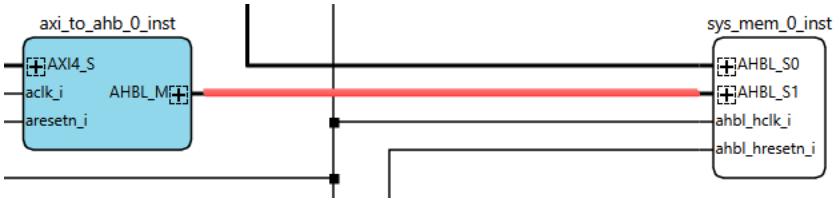


Figure 8.2. AXI to AHB Bridge

If the IP controller uses an AXI interface, it can be connected to the existing AXI interconnect block available in the design, as illustrated below.

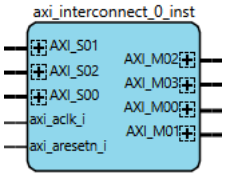


Figure 8.3. AXI Interconnect Instance

Add necessary input and output ports for the peripheral's signals, connect them in the design, and assign them to their respective ball locations in the Physical Design Constraints (.pdc) file based on the board specification.

8.1.2. Update the RISC-V C Application

8.1.2.1. Add New Mailbox Register

To enable the host application to communicate with the newly added peripheral controller, a new mailbox register must be defined. To extend the mailbox interface with additional registers, open the mailbox.h header file in the C project within Propel. The file contains all currently defined mailbox registers (refer to Figure 8.4). New registers must be appended after the last existing entry.

```
/* Offsets (Sysmem 1 Memory Space: corrected, all 4-byte aligned) */
#define MAILBOX_OFF_CMD              (MAILBOX_BASE + 0x0000u)
#define MAILBOX_OFF_DST_LO           (MAILBOX_BASE + 0x0004u)
#define MAILBOX_OFF_DST_HI           (MAILBOX_BASE + 0x0008u)
#define MAILBOX_OFF_SRC_LO           (MAILBOX_BASE + 0x000Cu)
#define MAILBOX_OFF_SRC_HI           (MAILBOX_BASE + 0x0010u)
#define MAILBOX_OFF_TRIGGER          (MAILBOX_BASE + 0x0014u)
#define MAILBOX_OFF_FLASH_START_ADDR (MAILBOX_BASE + 0x0018u)
#define MAILBOX_OFF_LEN              (MAILBOX_BASE + 0x001Cu)
#define MAILBOX_OFF_OP_STATUS        (MAILBOX_BASE + 0x0100u)
#define MAILBOX_OFF_FLASH_WR_DATA    (MAILBOX_BASE + 0x1100u)
```

Figure 8.4. Existing Mailbox Register Definitions in mailbox.h Header File

Important: The MAILBOX_OFF_FLASH_WR_DATA register occupies a reserved 4 kB block used to stage bitstream data before SPI flash programming. To prevent data corruption, any new register must be assigned an offset of 0x1200 or higher.

The base addresses of all mailbox registers are derived from the System Memory #1 base address. If the base address of System Memory #1 is modified in the hardware design, the corresponding base address definitions in mailbox.h must be updated accordingly to ensure correct operation.

```
/* Sysmem 1 base (same as platform) */
#define MAILBOX_BASE                 (SYSMEM1_INST_BASE_ADDR)
```

Figure 8.5. Mailbox Base Address Definitions Derived from System Memory #1

8.1.2.2. Add New Mailbox Command

To introduce a new command to the mailbox interface, open mailbox.h in the C project within Propel. The file lists all currently supported commands (refer to Figure 7.6). Append the new command definition after the existing entries.

```
/* Command codes (plan) */
#define MAILBOX_CMD_4K_FLASH_WRITE    (1u)
#define MAILBOX_CMD_REFRESH           (2u)
#define MAILBOX_CMD_RETURN_256B_DATA  (3u)
```

Figure 8.6. Existing Mailbox Command Definitions in mailbox.h

Once the command has been defined in mailbox.h, open mailbox.c and locate the *mailbox_execute_command* function. Add a corresponding case entry within the switch statement to handle the new command (refer to Figure 8.7). This ensures that the RISC-V application can correctly identify and process the command at runtime.

```
switch (cmd) {
case MAILBOX_CMD_4K_FLASH_WRITE:
    err = cmd_4k_flash_write(&ret_len);
    break;
case MAILBOX_CMD_REFRESH:
    err = cmd_refresh(&ret_len);
    break;
case MAILBOX_CMD_RETURN_256B_DATA:
    ret_len = 256;
    break;
default:
    mailbox_write32(MAILBOX_OFF_OP_STATUS, MAILBOX_OP_STATUS_FAIL);
    ret_len = 4u;
    err = -1;
    break;
}
```

Figure 8.7. Switch Statement in mailbox_execute_command() Handling Defined Commands

If the command does not require a response to host, omit *mailbox_set_gpio_return* function execution for the newly added command.

After updating the code, compile the C application as described in the Creating and Compiling the C Project in Propel section.

8.1.3. Compilation

Once the C application has been successfully compiled, refer to the Generating the Bitstream File section for instructions on updating the System Memory with the new application .mem file and recompiling the design. Following that, refer to the Programming the Bitstream File section to program the updated bitstream into the FPGA device.

8.2. Implement AI Inference

This section describes how to extend the reference design to enable the RISC-V soft processor on the FPGA to run inference engine such as TensorFlow Lite (TFLite) on data received from the host and return the result back over the PCIe interface. Follow the steps below to implement this capability.

Important: The RISC-V soft processor in this design does not include a hardware floating-point unit. Models must therefore be fully quantized to INT8 prior to deployment. Floating-point and fixed-point (non-quantized) models are not supported and will not execute correctly on this platform.

8.2.1. Update Propel Builder Design

Begin by removing IP blocks that are unnecessary for the AI inference use case to free up FPGA fabric resources – particularly LUTs and embedded memory – for the AI inference engine. The following blocks may be safely removed: the Octal SPI IP block, the FPGA Config block, and the 7-segment LED Control block (if the 7-segment display is not used on the target board).

Next, increase the size of System Memory 0, which serves as the program memory for the RISC-V application, to accommodate the AI inference engine and its associated libraries, which have a significant memory footprint. The default size is 64 kB; increase this to a value appropriate for the target model size and runtime requirement.

System Memory 1 acts as the shared data buffer between the PCIe interface and the RISC-V application. This is where the host deposits inference input data and where the RISC-V processor reads from and writes results back to. Increase the size of this System Memory if the expected inference payload size exceeds the current allocation.

To optimize inference performance, the Lattice CNN Coprocessor Unit may be leveraged as a dedicated hardware accelerator.

8.2.2. Update RISC-V C Application

8.2.2.1. Add New Mailbox Register

To designate a memory region within System Memory 1 for holding inference input data sent by the host, define a new mailbox register macro named MAILBOX_OFF_INFERENCE_DATA in the mailbox.h header file. Append this entry after the existing register definitions shown in Figure 8.8. This offset must be assigned an offset of 0x1200 or higher to avoid overlapping with the reserved 4 kB flash write staging area. Writing to this region corrupts any active bitstream update operation.

```
#define MAILBOX_OFF_CMD             (MAILBOX_BASE + 0x0000u)
#define MAILBOX_OFF_DST_LO          (MAILBOX_BASE + 0x0004u)
#define MAILBOX_OFF_DST_HI          (MAILBOX_BASE + 0x0008u)
#define MAILBOX_OFF_SRC_LO          (MAILBOX_BASE + 0x000Cu)
#define MAILBOX_OFF_SRC_HI          (MAILBOX_BASE + 0x0010u)
#define MAILBOX_OFF_TRIGGER         (MAILBOX_BASE + 0x0014u)
#define MAILBOX_OFF_FLASH_START_ADDR (MAILBOX_BASE + 0x0018u)
#define MAILBOX_OFF_LEN             (MAILBOX_BASE + 0x001Cu)
#define MAILBOX_OFF_OP_STATUS       (MAILBOX_BASE + 0x0100u)
#define MAILBOX_OFF_FLASH_WR_DATA   (MAILBOX_BASE + 0x1100u)
#define MAILBOX_OFF_INFERENCE_DATA  (MAILBOX_BASE + 0x1200u)
```

Figure 8.8. Appending MAILBOX_OFF_INFERENCE_DATA to the Mailbox Register List in mailbox.h

8.2.2.2. Add New Mailbox Command

Define a new command, MAILBOX_CMD_START_INFERENCE, by appending it to the existing command list in mailbox.h as shown in Figure 8.9. This command is the signal the host sends to instruct the RISC-V application to begin inference processing on the data already deposited into MAILBOX_OFF_INFERENCE_DATA.

```
#define MAILBOX_CMD_4K_FLASH_WRITE    (1u)
#define MAILBOX_CMD_REFRESH           (2u)
#define MAILBOX_CMD_RETURN_DATA       (3u)
#define MAILBOX_CMD_START_INFERENCE   (4u)
```

Figure 8.9. Appending MAILBOX_CMD_START_INFERENCE to the Mailbox Command List in mailbox.h

Once the command is defined, open mailbox.c and locate the *mailbox_execute_command* function. Add a corresponding case entry within the switch statement to handle MAILBOX_CMD_START_INFERENCE. Within this case handler, retrieve the inference input data from MAILBOX_OFF_INFERENCE_DATA and pass it as the input to the interpreter for inference execution.

Once the interpreter returns the inference result, write the output data to the MAILBOX_OFF_OP_STATUS register. Then, assign the byte length of the response to the ret_len variable. This value is subsequently written to the GPIO register, which signals the PCIe System Memory Bridge module of the exact number of bytes to transfer upstream to the host. The host receives this data as part of the operation completion interrupt flow described in Figure 3.5.

After completing all code modifications, recompile the C application following the procedure described in the Creating and Compiling the C Project in Propel section.

8.2.2.3. Compilation

Once the C application has been successfully compiled, follow the procedure in the Generating the Bitstream File section to update the System Memory component with the newly generated .mem file and regenerate the FPGA bitstream. Note that any changes to the System Memory sizes in the Update Propel Builder Design section must also be reflected in the Propel Builder design before regenerating. Thereafter, refer to the Programming the Bitstream File section to program the updated bitstream onto the FPGA device.

8.2.2.4. Host Application

On the host side, implement the host application following the operational flow illustrated in Figure 8.10. This flowchart outlines the sequence of steps required for the host to send inference input data to the FPGA, trigger the inference operation, and retrieve the result upon completion.

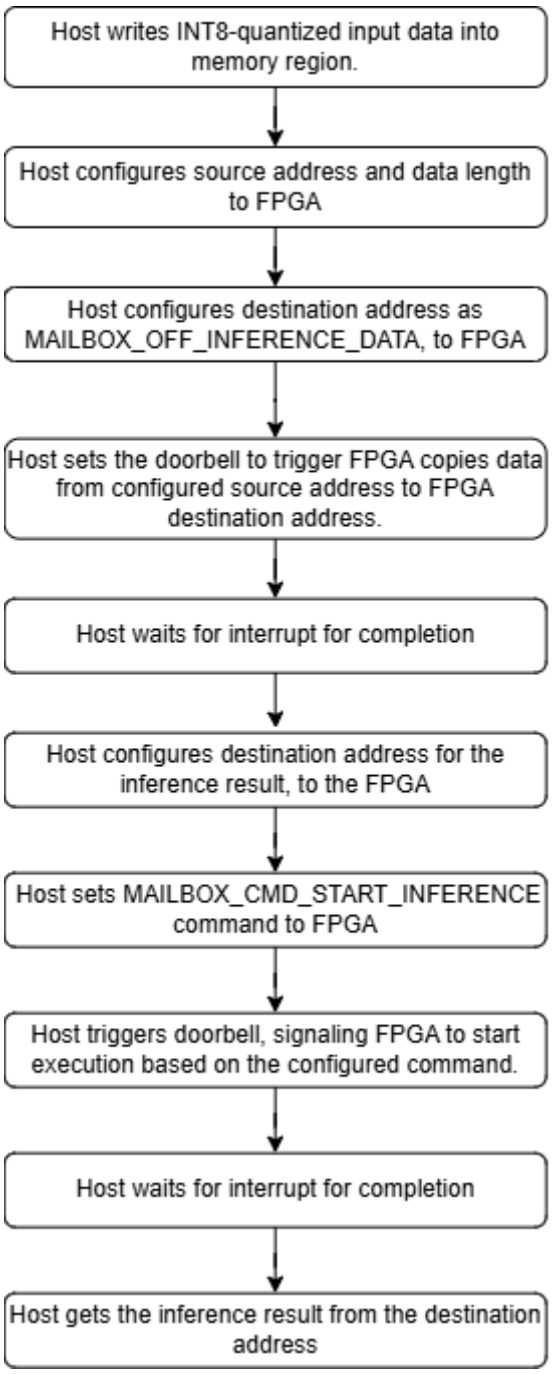


Figure 8.10. Flowchart of Host Application Flow for Triggering and Retrieving AI Inference Results

References

- CNN Coprocessor Unit IP Core User Guide (FPGA-IPUG-02170)
- Octal SPI Controller IP User Guide (FPGA-IPUG-02273)
- PCIe x1 IP Core User Guide (FPGA-IPUG-02091)
- PCIe x4 IP Core User Guide (FPGA-IPUG-02126)
- Lattice Solutions Reference Designs web page
- Lattice Solutions IP Cores web page
- Lattice Propel Design Environment web page
- Lattice Radiant Software User Guide
- Lattice Radiant FPGA design software
- Lattice Insights for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

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

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

Revision History

Revision 1.0, September 2026

Section	Change Summary
All	Initial release.

www.latticesemi.com