Lattice sensAI Neural Network Compiler Software

User Guide

FPGA-UG-02052-8.0

December 2025

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.

Contents

Contents 3
Abbreviations in This Document 8
1. Introduction 9
1.1. Prerequisites 9
1.2. Purpose 10
1.3. Limitations 10
2. Installing the Software 11
3. Getting Started 14
3.1. Creating a New Project 14
3.2. Opening an Existing Project 18
3.3. Saving a Project 19
3.4. Inputs 19
3.5. Help 20
3.6. Command Line Interface 20
3.7. Design Restrictions 23
3.8. Next Steps 27
4. Working with Projects 28
4.1. Implementations 28
4.2. Project Flow 29
4.3. Views 35
4.4. Example Projects 39
5. Advanced Topics 45
5.1. Project Implementation Settings 45
5.2. Quantization 58
5.3. Optimization Modes 69
5.4. SensAI Security Flow 69
6. Supported Frameworks 71
6.1. Caffe 71
6.2. TensorFlow 71
6.3. Keras 76
7. SB Debugging 82
7.1. Hardware Configuration 82
7.2. Debug Window Options 88
7.3. Driver Installation 89
7.4. USB Debugging API Interface 90
7.5. Board Detection Troubleshooting 92
7.6. CrossLink-NX, CertusPro-NX and Avant Layer by Layer USB Debug 93
8. RISC-V Register Interface Generator 98
8.1. Launch RISC-V System Generator Environment 98
8.2. Generate CSR Register IP Cores 99
Appendix A. Supported Keras Layers 104
Appendix B. Supported Layer Configuration 106
Appendix C. Supported TensorFlow Operations 108
Appendix D. USB Debugging Register Map 111
Appendix E. Supported ONNX Layers 112
Appendix F. Network Topology and Device Table 113
Appendix G. Common CNN Blocks Used in Lattice NNC 114
References 127
Technical Support Assistance 128
Revision History 129

Figures

Figure 2.1. Installation Location Specification 11
Figure 2.2. Installation Component Specification 12
Figure 2.3. Installation Ready to Install Dialog Box 12
Figure 2.4. Lattice Neural Network Compiler Software for Windows Splash Screen 13
Figure 3.1. Project Settings Window 14
Figure 3.2. Example cmd for Post Processing 14
Figure 3.3. Proto File Selection Window 15
Figure 3.4. Project Implementation Options Window 16
Figure 3.5 Project Implementation Window 2 (Only for Advanced IP) 17
Figure 3.6. Project Window 18
Figure 3.7. Load Project Window 18
Figure 3.8. Python Code for Raw Input 19
Figure 3.9. Multiple Input Selection Window 19
Figure 4.1. Project Implementation Options Window 28
Figure 4.2. Analyze Results 29
Figure 4.3. Compile Results 32
Figure 4.4. Simulate Results 33
Figure 4.5. Data Histogram for the Blob 34
Figure 4.6. Post Processing 34
Figure 4.7. Input Network – TensorFlow or Keras 35
Figure 4.8. Close Tensorboard Process 36
Figure 4.9. GUI Themes 36
Figure 4.10. HTML Log 37
Figure 4.11. Default View of HTML log 37
Figure 4.12. Search Functionality of Warning 38
Figure 4.13. Simulation Data Graph 38
Figure 5.1. Project Implementation Window – ECP5 45
Figure 5.2. Project Implementation Window – UltraPlus (1) 46
Figure 5.3. Project Implementation Window – UltraPlus (2) 46
Figure 5.4. Project Implementation Window – CrossLink-NX-Optimized 47
Figure 5.5. Project Implementation Window – CrossLink-NX-Compact 47
Figure 5.6. Project Implementation Window – CertusPro-NX-Optimized 48
Figure 5.7. Project Implementation Window – CertusPro-NX-Compact 48
Figure 5.8. Project Implementation Window – CertusPro-NX-Extended 49
Figure 5.9 Project Implementation Window – CertusPro-NX Advanced IP Part 1 49
Figure 5.10 Project Implementation Window – CertusPro-NX Advanced IP Part 2 50
Figure 5.11 Project Implementation Window – Avant Advanced IP Part 1 50
Figure 5.12 Project Implementation Window – Avant Advanced IP Part 2 51
Figure 5.13. On-the-Fly Post Processing Format 55
Figure 5.14. On-the-Fly Post Processing Data Flow 55
Figure 5.15. Create Quantized Version Flag 61
Figure 5.16. Tensor Graph Quantization Nodes 65
Figure 5.17. Activation Data Quantization Nodes 68
Figure 5.18. SensAI Security Flow: Encrypt Model 69
Figure 5.19. SensAI Security Flow: Encrypted Model Selection 70
Figure 5.20. SensAI Security Flow: Encrypt Model 70
Figure 6.1. Original TensorFlow Training Model 72
Figure 6.2. Simplified TensorFlow Inference Model 73
Figure 6.3. Tensorboard Visualization of Binarization 75
Figure 6.4. Binary Neural Network Modes in TensorFlow 76
Figure 7.1. Cypress Window 83

Figure 7.2. Radiant Programmer – Default Screen 84
Figure 7.3. Radiant Programmer Device Selection 84
Figure 7.4. Radiant Programmer – Device Operation....84
Figure 7.5. Selecting Device Properties for CrossLink-NX....85
Figure 7.6. Output Console after Successful Flashing....86
Figure 7.7 Avant Board with FX3 USB Board 87
Figure 7.8. USB Debug Window....88
Figure 7.9. USB3-GigE VIP Board Label 92
Figure 7.10. CNX-VnV Board Label....92
Figure 7.11. CPNX-VnV Board Label....92
Figure 7.12. USB Debug Window....93
Figure 7.13. USB Debug Firmware Generation....94
Figure 7.14. Upload FW, Input and Run USB-Debugging....94
Figure 7.15. Read USB Data with Blob Selected 95
Figure 7.16. Read USB Data without Blob Selected....95
Figure 7.17. Save USB Data....96
Figure 7.18. Expected Values for Corresponding Blob 96
Figure 7.19. Show Expected vs HW MAE 97
Figure 8.1. Opening the RISC-V System Generator....98
Figure 8.2. System Generator Home Window 99
Figure 8.3. System Generator Functions....99
Figure 8.4. System Generator Add New Register 99
Figure 8.5. System Generator Add and Remove Register Field 100
Figure 8.6. System Generator Register Bit Width Limitation....100
Figure 8.7. System Generator Example CSR Template 101
Figure 8.8. CSR Register Example....101
Figure 8.9. System Generator Save Project 102
Figure 8.10. System Generator Generate IPK File 102
Figure B.1. Sigmoid Function 105
Figure B.2. Strided Slice Example....105
Figure C.1. Batch Normalization 108
Figure C.2. Unpool Implementation 110
Figure H.1. Non-Quantized 3x3 CBSR or 3x3 Depthwise CBSR 114
Figure H.2. Quantized 3x3 CBSR or 3x3 Depthwise CBSR 115
Figure H.3. Non-Quantized 1x1 CBSR 115
Figure H.4. Quantized 1x1 CBSR 116
Figure H.5. Non-Quantized Add Block 116
Figure H.6. Quantized Add Block 117
Figure H.7. VGG toy model 117
Figure H.8. MobileNetV1 Block....118
Figure H.9. MobileNetV1 Toy Model 118
Figure H.10. MobileNetV2 Block 1....119
Figure H.11. MobileNetV2 Block 2....119
Figure H.12. ResNet Toy Model 120
Figure H.13. ResNet Block 2 Variation 1 120
Figure H.14. ResNet Block 2 Variation 2 121
Figure H.15. ResNet Block 2 Variation 3 122
Figure H.16. GoogleNet Inception Block 1....123
Figure H.17. GoogleNet Inception Block 2....123
Figure H.18. Init Block....124
Figure H.19. DownSample Block....124
Figure H.20. Regular Block....125

Figure H.21. Upsample Block 125

Tables

Table 3.1. Arguments and Usage 20
Table 5.1. Learned Step Quantization Details with Device Type 61
Table 5.2. Unsigned 8-Bit Quantization (Fixed Point Quantization) 63
Table 5.3. Signed 8-Bit Quantization (Fixed Point Quantization) 63
Table 5.4. Fixed Point Quantization Details with Device Type 63
Table 5.5 Quantization Support in Layers 64
Table 5.6. SensAI Security Flow: File Extension Mapping 70
Table B.1. Supported Layer Configuration 106
Table E.1. USB Debugging Register Map 111
Table F.1. Network Topology and Device 113
Table G.1. Enet Example Architecture 126

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
BNN	Binarized Neural Networks
CLI	Command Line Interface
CNN	Convolutional Neural Network
CNX	CrossLink-NX
CPNX	Certus-Pro-NX
CSR	Control and Status Register
DRAM	Dynamic Random Access Memory
FC	Fully Connected
FPQ	Fixed Point Quantization
FPS	Frames Per Second
GUI	Graphic User Interface
HRAM	Hyper Random-Access Memory
IP	Intellectual Property
LRAM	Large Random-Access Memory
LSQ	Learned Step Quantization
LUT	Lookup Table
ML	Machine Learning
NCHW	Number of Samples, Channels, Height, Width
NNC	Lattice Neural Network Compiler tool
ONNX	Open Neural Network Exchange
PTQ	Post Training Quantization
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RTL	Register Transfer Level
TCL	Tool Command Language
USB	Universal Serial Bus

1. Introduction

This document describes the usage and troubleshooting of Lattice Neural Network Compiler software.

1.1. Prerequisites

The hardware, software, connection, and general requirements for this demonstration are provided in the following sections.

1.1.1. Hardware Requirements

The software requires the following hardware components:

- PC with either Windows 10 x64 or newer; or PC with compatible Ubuntu x64 distribution for running software flow only.
- Lattice Inference Machine-compatible FPGA.

1.1.2. Software Requirements

This software product requires the following software components:

- Lattice Neural Network Compiler Software for Windows or Linux.
- Diamond Programmer System software for downloading the FPGA bitstream.
- Lattice Diamond™ design software for modifying the platform and regenerating the bitstream.
- Radiant Programmer System software for downloading FPGA bitstream.
- Lattice Radiant™ design software for modifying the platform and regenerating the bitstream.

1.1.3. Connection Requirements

Programming the device and running Lattice Neural Network Compiler Software directly from the GUI requires a Windows installation and a Windows-compatible connection, such as the USB driver for Lattice FPGA development boards.

1.1.4. General Requirements

This document requires knowledge of the following:

- Familiarity with TensorFlow or Keras machine learning frameworks.
- Familiarity with Lattice FPGA development, including basic concepts and troubleshooting skills, as well as experience establishing connectivity between the device and computer or using other hardware (such as an SD card) for transfer data to the target hardware.

1.1.5. IP Requirements

- Neural Network Compiler 8.0 supports IP cores for the ECP5, iCE40 UltraPlus, CrossLink-NX, CertusPro-NX, and Lattice Avant device families.
- For ECP5, use CNN Accelerator IP Core v2.1.
- For iCE40 UltraPlus, use Compact CNN Accelerator IP Core v2.0.0.
- For CrossLink-NX, use Crosslink-NX CNN Accelerator IP Core v3.0.
- For CertusPro-NX, use CertusPro-NX CNN Accelerator IP Core v3.0.
- For Lattice Avant, use Advanced CNN Accelerator IP Core v3.0.
- IP cores from previous releases may not work with this version. Ensure you use the versions provided by Lattice for Neural Network Compiler 8.0.

1.2. Purpose

This application shows the ability and features of Lattice Neural Network Compiler Software to:

- Analyze and compile a neural network for use with selected Lattice Semiconductor FPGA products.
- Simulate hardware to obtain expected fixed and floating-point output.
- Download and run neural networks directly on hardware via USB debugging.
- Manage multiple implementations per project to view the effects of different strategies.

1.3. Limitations

The following cautions apply to the software as a whole:

- Operations are conducted in fixed point notation on the hardware as a result of floating-point values being converted to and from fixed point representation.
- Specific neural network features, such as layers or functions, require certain configurations to function or may not be supported.

2. Installing the Software

The demonstration package of the Lattice Neural Network Compiler Software is available as an executable installer for Windows and Linux. Install the software on Windows using the Machine Learning Software Setup executable (.exe) or on Ubuntu Linux using the run file (.run). Launch the installation process and customize the options as detailed in this section.

To install the Lattice Neural Network Compiler Software:

1. Close all applications before starting the Lattice Neural Network Compiler Software installation.
2. Double-click the Lattice Neural Network Compiler Software installer you downloaded.
3. The Welcome to Lattice Machine Learning Software 8.0 Software Setup dialog box opens.
4. Click **Next** to select the installation folder.
5. On Windows, the default destination folder is C:\lscc\ml\8.0; on Linux, it is ~/lscc/ml/8.0. Click **Browse** to change the destination (see Figure 2.1).

Figure 2.1. Installation Location Specification

6. Click **Next** to open the Product Options dialog box (see Figure 2.2).
7. Select the Machine Learning Software components you want to install by checking or clearing the listed options.

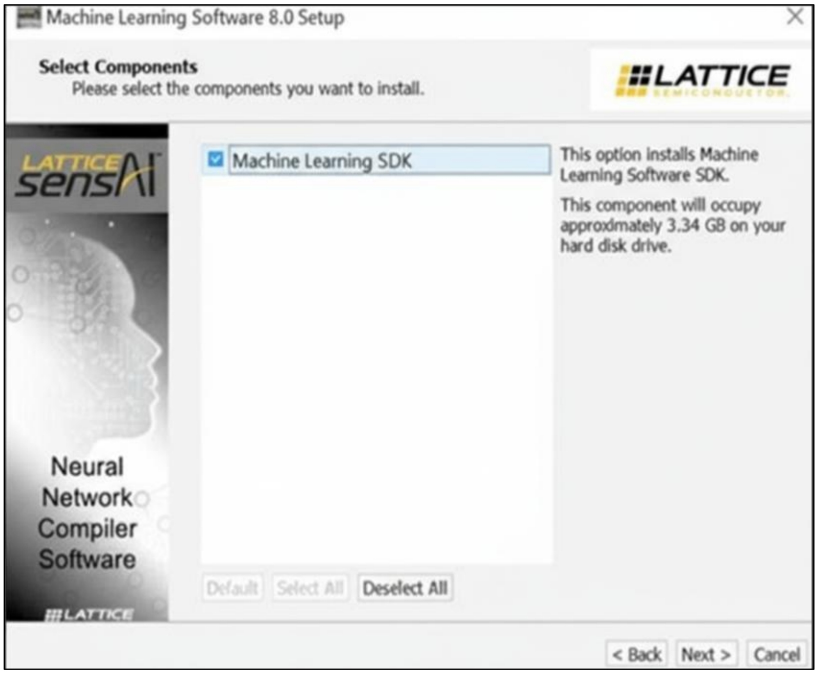


Figure 2.2. Installation Component Specification

8. Click **Next** to open the License Agreement dialog box.
9. Read the license agreement. If you agree, click **I accept the license** to open the Start Menu shortcuts dialog box.
10. Click **Next** to open the Select Program Folder dialog box. The default name is Lattice Machine Learning Software 8.0. If you want to change the name, enter it in the Program Folder text box.
11. Click **Next** to display the Ready to Install dialog box (Figure 2.3). Review the current settings, including the destination folder and selected components. If everything is correct, select **Install** to start the installation.

Figure 2.3. Installation Ready to Install Dialog Box

12. In the Installation Wizard Complete dialog box, read the confirmation note and click **Finish**.
13. Run the executable either by using the desktop shortcut or Start menu shortcut, if created, or by navigating to your installation directory and running **lsc_ml_compl.exe** on Windows or **lsc_ml_compl** on Ubuntu Linux. You can then see the main window, as shown in Figure 2.4.

Figure 2.4. Lattice Neural Network Compiler Software for Windows Splash Screen

14. The installed software is now ready for use.

3. Getting Started

In this chapter, you can learn how to use Lattice Neural Network Compiler Software to create new projects and edit existing projects.

3.1. Creating a New Project

A project is a collection of all the files necessary to create and download your design to the selected device. The New Project window guides you through the steps of specifying a project name and adding existing sources to the new project.

To create a new project:

1. From the main window, click **File > New**. The Project Settings window opens, as shown in Figure 3.1.

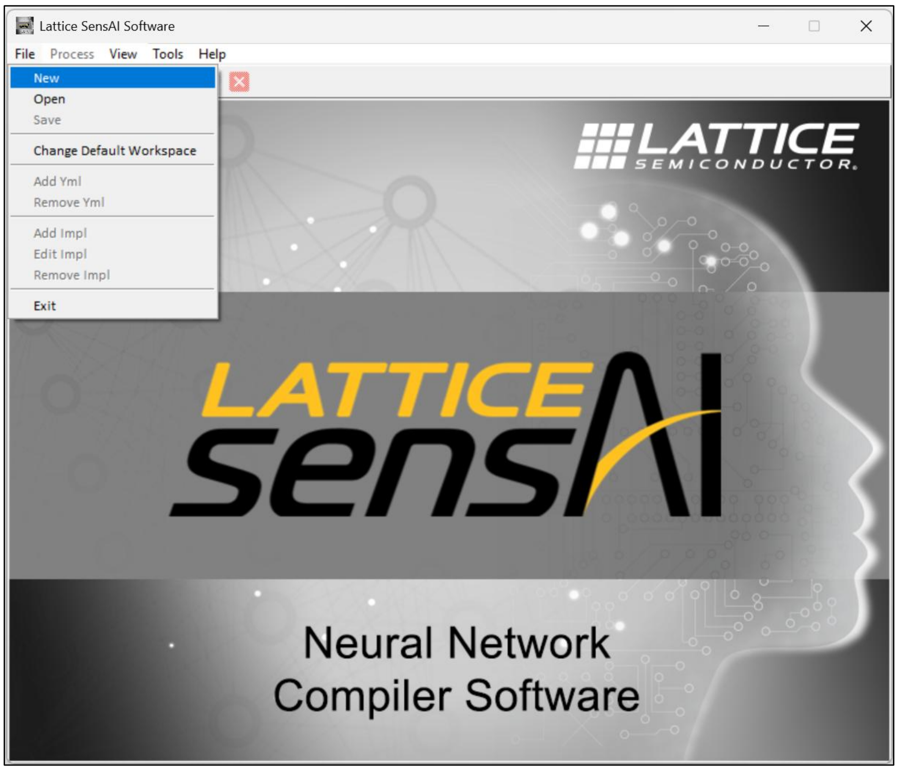


Figure 3.1. Project Settings Window

2. Enter a project name in the Project field at the top-left.
3. Select a framework for your design. Currently, sensAI™ supports TensorFlow, Keras, and ONNX (experimental).
4. Select the device you intend to run this network on.
5. Enter an optional post-processing command. Post-processing commands use the following format:

```
python test.py [<script-arg1>  <script-arg2> … ] <input-data-file> <simulation-npy-data-file>
```

Figure 3.2. Example cmd for Post Processing

The input-data-file and simulation-npy-data-file arguments displayed in the angle brackets are added by the sensAI tool in this command.

The script-arg parameters displayed in square brackets [ ] are script-dependent arguments.

1. Select a class for your network. SensAI supports Convolution Neural Network (CNN) and Binary Neural Network (BNN).
2. Select the MOBILENET mode checkbox if you want to use a model with the Mobilenet IP for ECP5 devices using the CNN class. See the Advanced Topics section for more information on Mobilenet mode. Similarly, select Compact mode, Optimized mode, or Extended mode from the drop-down list if you want to use a model for the respective IPs of the CrossLink-NX device and the CertusPro-NX device.
3. Click **Network File**. The Proto File Selection window opens, as shown in Figure 3.3.

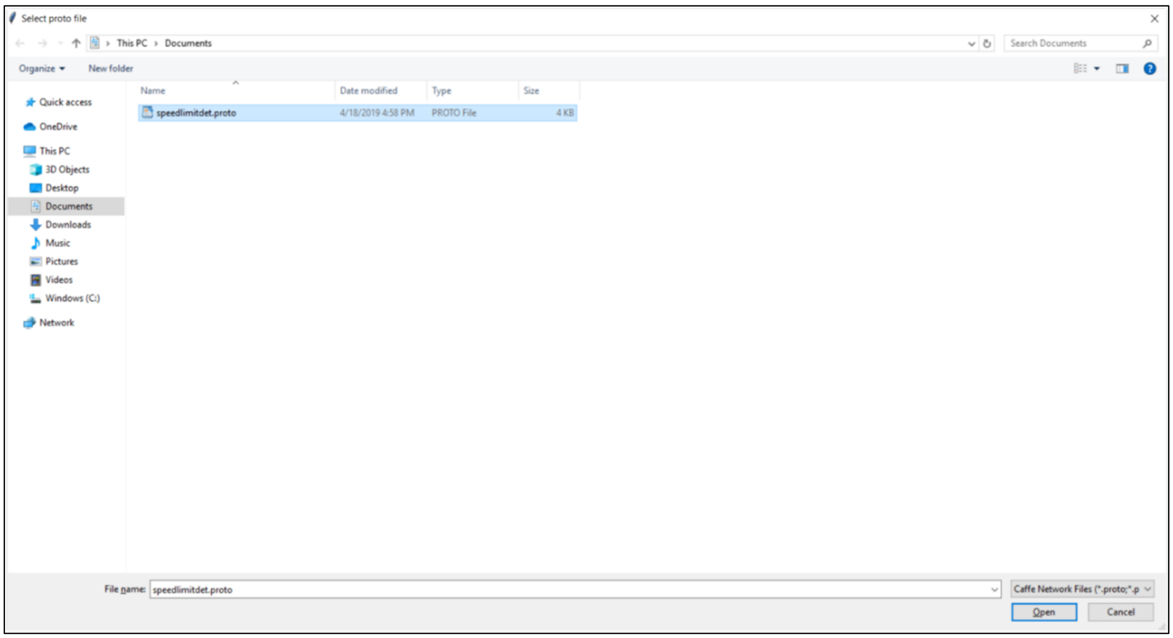


Figure 3.3. Proto File Selection Window

4. Navigate to your proto file and select it in the window.
5. Click **Open** to load the proto file into your project.
6. Click Model File and follow a similar process to steps 3-5, selecting your model file this time.
7. Click Image/Video Data and follow a similar process to steps 3-5, this time selecting your image or video file. You can check Scan Data Layer to let the software attempt to locate your data file if it is defined in your network.
8. Click **Next** to open the Project Implementation Options window, as shown in Figure 3.4.

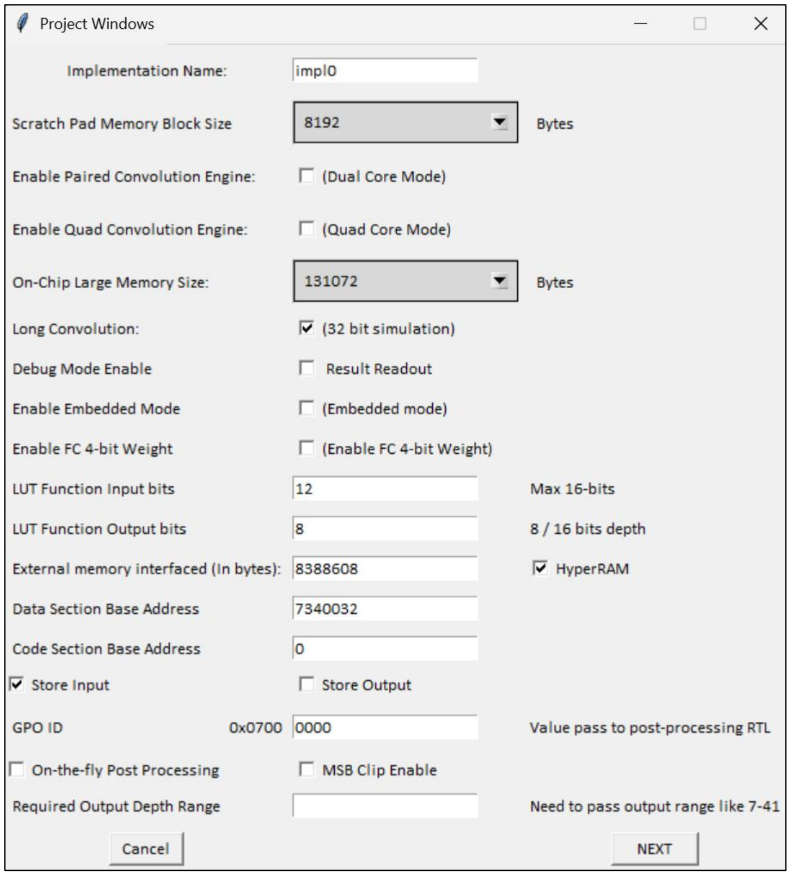


Figure 3.4. Project Implementation Options Window

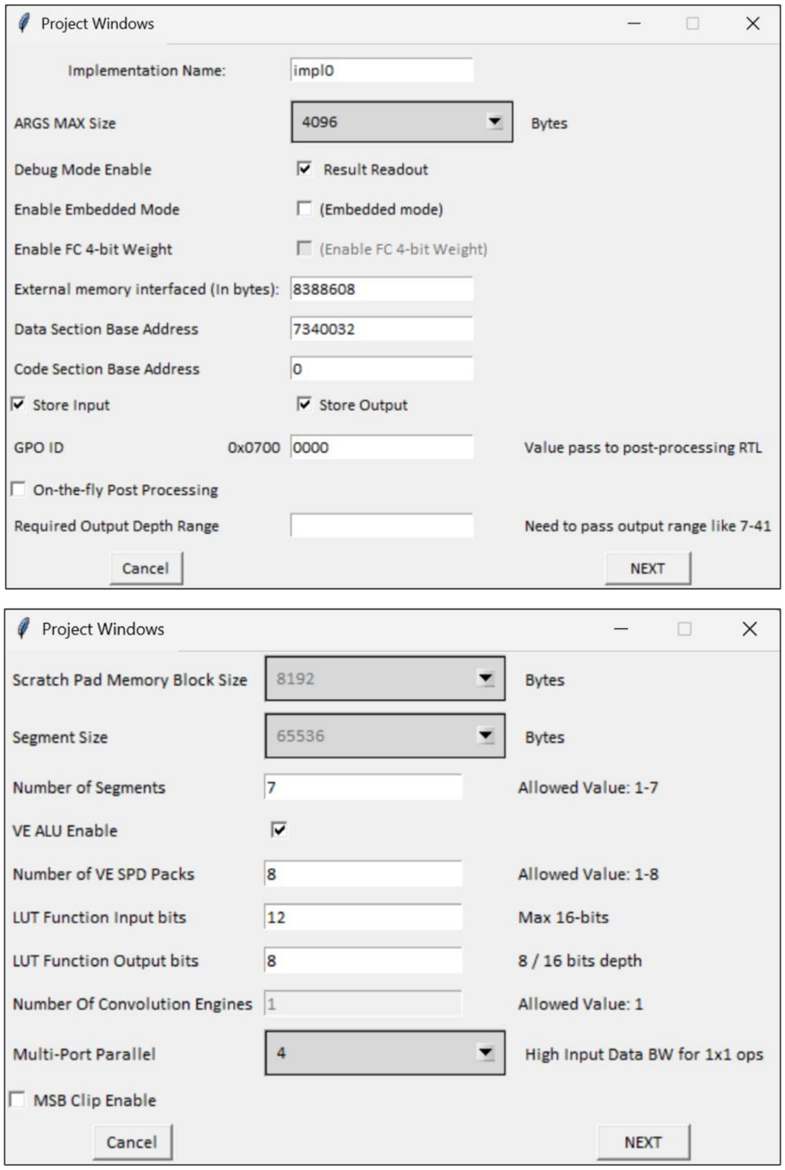


Figure 3.5 Project Implementation Window 2 (Only for Advanced IP)

9. The Project Implementation window is automatically filled with default settings for the Implementation Name, as well as the parameters. You can change the name and parameters if desired. For more information on how each parameter works and their limitations, read the Project Implementation Settings section.
10. Click **Ok** to create your project. The Project Window opens, as shown in Figure 3.6.

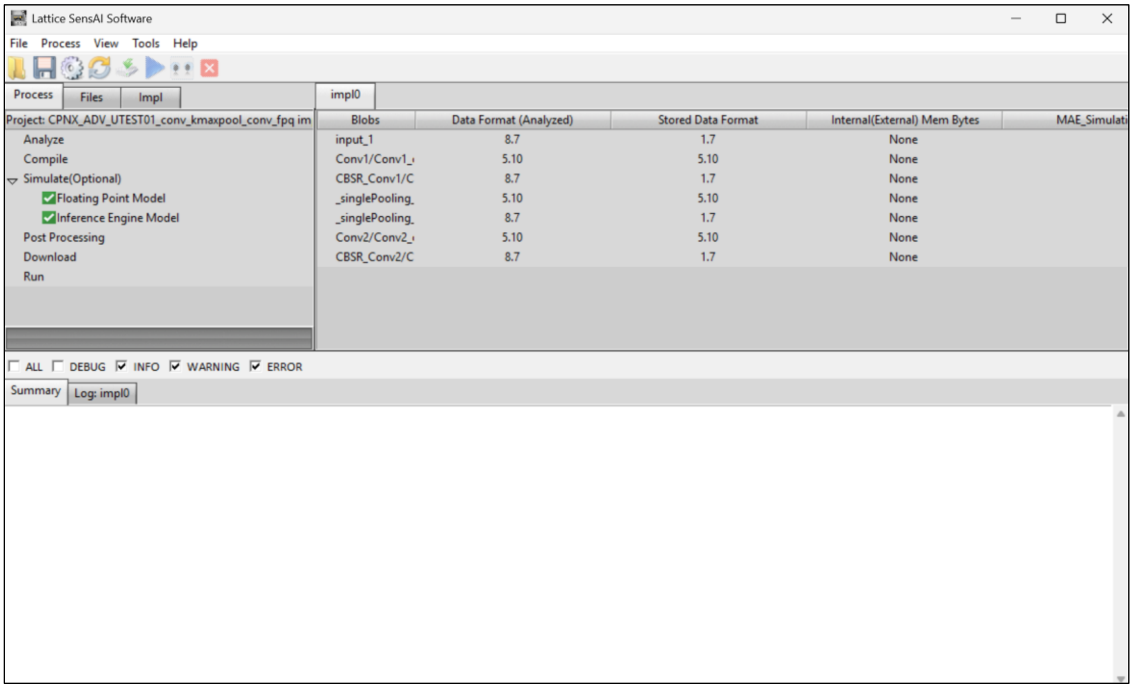


Figure 3.6. Project Window

3.2. Opening an Existing Project

1. Use one of the following methods to open an existing Lattice Neural Network Compiler Software project:
   - In the Main Window, click the **Open Project** button.
   - From the **File** menu, choose **Open**.

   The Open Project Window opens, as shown in Figure 3.7.

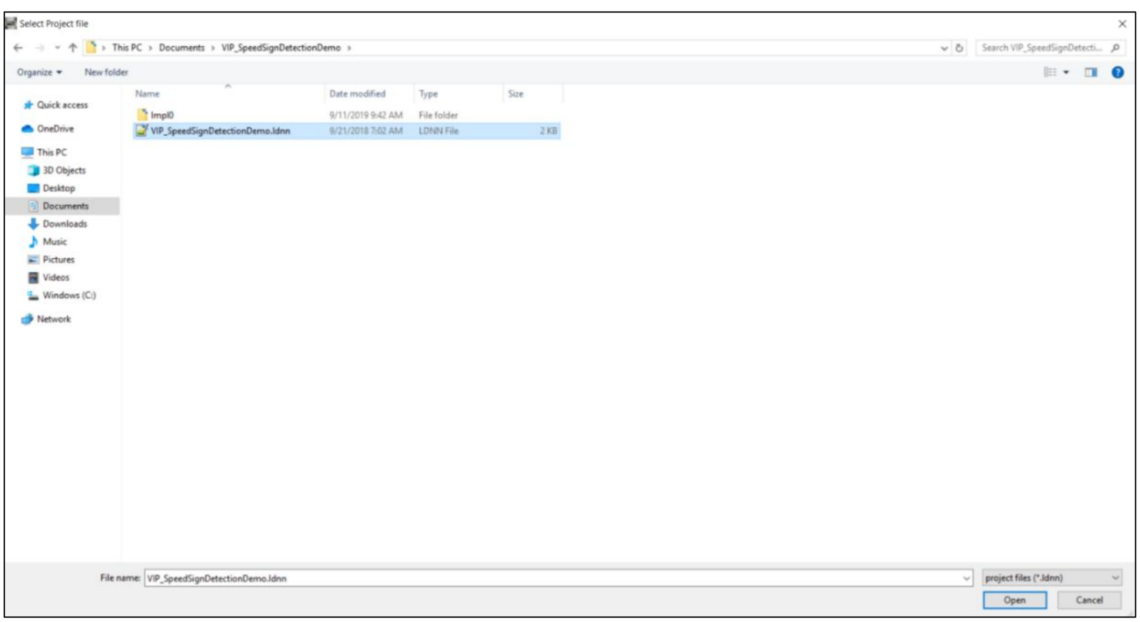


Figure 3.7. Load Project Window

2. Navigate to an existing LDNN type file and select it.
3. Click **Open** to open the project.

3.3. Saving a Project

When working on a project you want to save, click on the floppy disk icon or navigate to **File > Save** in order to save your project. This can save the files with the project name into the project directory, as specified in your project settings.

3.4. Inputs

In addition to images, sensAI supports other types of input data as well.

3.4.1. Audio Input

The tool only accepts .wav files with a minimum length of 1 second. There is no preprocessing performed on audio input as of version 7.0.

3.4.2. Raw Input

By enabling the **Raw Input** option when creating a new project, you can pass input data in the form of .npy array. The array size should match exactly with the inputs in the network. This is because the array is directly fed to the network without performing any preprocessing. For example, mean and scale are not used on raw input data. Preprocessing can be performed in Python and then passed as a saved numpy array to sensAI.

To save an array, ***A***, in a file, ***raw_input.npy***, it only requires two lines of Python code, as shown in Figure 3.8.

```
import numpy as np
np.save("raw_input.npy",A)
```

Figure 3.8. Python Code for Raw Input

Note: For image input as raw input, the data must be in BGR format.

3.4.3. Multiple Input Selection

The tool automatically detects if the model has multiple inputs. Select the image or raw input according to the model inputs. Model input names are displayed so you can select the input files accordingly. Figure 3.9 shows the input selection window.

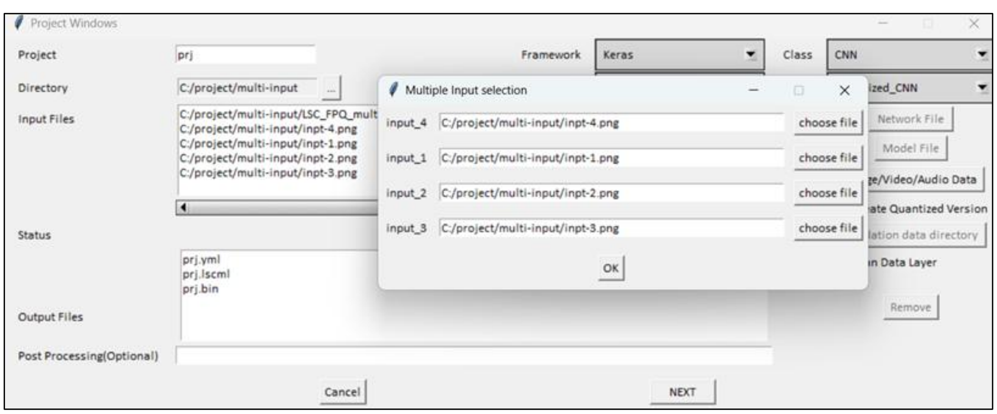


Figure 3.9. Multiple Input Selection Window

3.5. Help

For more software help, the **Help** menu contains links to relevant help topics.

3.5.1. About

To find out more versions and license information, navigate to **Help > About** to bring up the About window, which has tabs for different software information sections. The **About** tab contains information about the software. Your current version and build number are displayed here. The **License** tab provides a convenient way to view the license agreement.

3.5.2. User Guide

This user guide is routinely updated and may not be the latest version. To quickly go to the Lattice Semiconductor web page, which contains the latest version of the User Guide as well as supplemental material, navigate to **Help > User Guide,** and you will be taken to the correct page.

3.6. Command Line Interface

The executable can be used from a command line interface if you prefer not to use the GUI. To execute a command, launch the executable from the command line and pass it the arguments you wish to use.

For example, to bring up the help Windows CLI in Cygwin, the command is:

```
lsc_ml_compl.exe --help
```

While on Linux, execute it as:

```
./lsc_ml_compl --help
```

This brings up the help menu for the CLI. You can see the usage and arguments in the following sections of this chapter.

3.6.1. Arguments and Usage

Table 3.1. Arguments and Usage

Argument	Description
--h, --help	Show this help message and exit
--cryptography	To run encryption/decryption flow
--input_file_path	Input model path that user wants to Encrypt/Decrypt
--input_file_path	Output model path to store encrypted/decrypted model
--password	Password to perform encryption/decryption
--mode	To select mode from encrypt/decrypt
--gui [GUI]	Invoke GUI tool
--cmd [CMD]	Valid commands are analyze, compile, simulate, download, run, and all
--framework {TensorFlow,Keras,ONNX}	Framework used to train the network. Currently TensorFlow, Keras, and ONNX are supported. The support for Caffe has been removed.
--network_file NETWORK_FILE	• TensorFlow .pb file • Keras .h5 file • ONNX .onnx file
--image_files IMAGE_FILES	.jpg Image file
--num_conv_eng NUM_CONV_ENG	Number of convolution engines used. Only for CPNX and AVANT devices with Advanced CNN IP 4*N number of output channels are getting generated in parallel. N = 1 for CPNX and N = 1-4 for AVANT devices
--num_ebr NUM_EBR	Number of embedded block ram.
--ebr_blk_size {16384,32768,65536}	Size of each embedded block ram for UltraPlus.
--crosslink_scratch_pad_blk_size {1024,2048,4096,8192}	CrossLink-NX and CertusPro-NX scratch embedded block RAM size.

Argument	Description
--crosslink_lram_size {65536,131072,262144}	CrossLink-NX and CertusPro-NX On-chip large RAM size.
--cross_link_external_mem_size CROSS_LINK_EXTERNAL_MEM_SIZE	CrossLink-NX and CertusPro-NX External memory (dram/hyper ram) interfaced size.
--crosslink_code_base_addr CROSSLINK_CODE_BASE_ADDR	CrossLink-NX and CertusPro-NX Code/Binary base address of external memory.
--crosslink_data_base_addr CROSSLINK_DATA_BASE_ADDR	CrossLink-NX and CertusPro-NX data base address of external memory.
--hyper_ram {0,1}	Use hyper RAM as external memory in CrossLink-NX or CertusPro-NX.
--extmem_start_addr EXTMEM_START_ADDR	Starting address of external DRAM to store data.
--mean MEAN	Mean value used to preprocess data during training.
--scale SCALE	Scale value used to preprocess data during training.
--sample_rate SAMPLE_RATE	Sample rate value used for sampling the audio file.
--down_sampling DOWN_SAMPLING	Down sampling value used for down sampling the audio file.
--extmem_off {0,1}	Turn off using external memory to store data. By default, external memory is used to store input/output and scratch data.
--load_from_extmem {0,1}	By default, data is loaded from external memory to internal memory. If this option is '0', it makes sure data is directly loaded to EBR from sensor or host.
--store_to_extmem {0,1}	By default, data is output to external memory. If this option is '0', it makes sure to read data from internal memory.
--project_name PROJECT_NAME	Sets the project name.
--project_dir PROJECT_DIR	Project Directory.
--device {Ultra Plus, ECP5, CrossLink-NX, CertusPro-NX, AVANT}	Sets the Device to ECP5, UltraPlus, CrossLink-NX, CertusPro-NX or Avant.
--mobilenet_mode {0, 1}	Enable MOBILENET mode by setting value to 1. Default is 0.
--ip_mode {Optimized_CNN, Compact_CNN, Extended_CNN, Advanced_CNN}	Sets the machine learning (ML) intellectual property (IP).
--nnMode {0,1}	Sets class CNN(0)/BNN(1).
--bnn_sign_mode {0,1}	Quantization mode for BNN(0: “0/1” and 1: “+1/-1”)
--enable_hw_sim {0,1}	Enable Hardware simulation. Default is 1.
--enable_fixed_sim {0,1}	Enable Fixed-point simulation. Default is 1.
--enable_float_sim {0,1}	Enable Floating-point simulation. Default is 1.
--collapse_layer {0,1}	Collapse layers. Default is 0.
--enable_dualcore {0,1}	Enable Dual core functionality. Default is 1.--enable_dualcore {0,1}
--enable_quadcore {0,1}	Enable Quad core functionality for CertusPro-NX Optimized Only. Default is 0.
--enable_embedded_mode{0,1}	Enable Embedded Mode. Default is 0.
--input_ebr INPUT_EBR	Specify comma separated input EBR numbers.
--output_ebr OUTPUT_EBR	Specify comma separated output EBR numbers.
--reg_out {0,1}	Enable Register out functionality for CrossLink-NX, CPNX and Avant device. Default: 0.
--required_output_depth_range REQUIRED_OUTPUT_DEPTH_RANGE	Specify Required Output Depth Range. For example, “7-13” only processes the 7th to 13th filters of the output convolution layer.
--user_added_yml USER_ADDED_YML	Specify User added yml file.
--conv1x1_mode {single,quad,dual}	Specify conv1x1 mode like quad, dual or single. Default single for iCE40 UltraPlus, CrossLink-NX and CertusPro-NX Compact.
--scratch_blk_size {1024,2048,4096,8192}	Size of scratch embedded block RAM for UltaPlus. Default: 8192.
--arg_max {4096, 8192}	Size of memory block RAM for arg max operation. Functionality for Extended and advanced CNN only. Default: 4096.
--otf_post_processing {0,1}	Specify on the fly post processing for UltraPlus. Default: 0.

Argument	Description
--number_of_det_class NUM_OF_DET_CLASS	Size of scratch embedded block RAM for UltaPlus. Default: 4096.
--enable_debug_mode {0,1}	Enable debug mode or not. Supported only in CNX, CPNX and Avant devices. Default: 0.
--num_ve_segments	LRAM Segment numbers we want to use . lram size will be equal to (number of segments x segment size) value ranges from 1 to 7 for CPNX Advanced CNN and 1 to 16 for Avant Advanced CNN IP. Default value : 16.
--segment_size	Size of segment for advanced CNN IP. For CPNX and Avant device, advanced IP, with 32 bit datapath size segment size has fixed value of 65536. For Avant device, for 64 bit datapath size, segment size is 131072.
--num_ve_spd_packs	Number of the VE scratchpad in advanced CNN IP. Values ranges from 1 to 8.
--ve_alu_enable	Enable VE ALU or not. For Advanced IP only.
--multi_port	Multi-Port Parallel Values for advanced CNN IP. Values : {2,4}.
--datapath_width {32, 64}	Width of datapath for transferring of data within IP. More datapath width means more bytes of data transferred in each transaction.
--lut_input_bits {5,6,7,8,9,10,11,12}	Input bits for LUT for activation function. Only available in Optimized_CNN.
--lut_output_bits {8, 16}	Output bits of data given by LUT of activation function. Only available in Optimized_CNN IP.
--msb_clip_enable {0,1}	Clip MSB of input data bit for LUT of activation function.
`--create_quantized_version {0, 1}`	Create a quantized version of the selected input model. If the input model is not quantized, enabling this creates a quantized version of the input model to be used for further network compilation processing. Default: 0.
--validation_data_path {path of directory}	Path of directory containing validation data. The compiler tool uses the validation data contained in this directory when creating the quantized version of a model.
--enable_fc_4_bit_weights {0, 1}	Enable weights of Fully Connected (FC) engine to be converted into 4 bits. Otherwise, used as 8 bits. Default: 0.
--number_of_ml_ips	Number of ML IP used to run the network. Default: 1.
--external_memory_port	Active only when LPDDR4 is selected. Based on the external memory port, logical external memory addresses are derived when the compiler generates instructions. This external memory port mapping must match with the hardware address mapping used in the RTL design.
Commands for Multi-Input Network	
--image_files “input1_name:IMAGE1_PATH; input2_name:IMAGE2_PATH”	Specify the input image names according to the input model. Separate input image names with the semicolon (;).
--multi_input_scale “input1_name:0.0078125;input2_name: 0.0078125”	Specify the different scale values for each input.
--multi_input_mean “input1_name:1; input2_name: 1”	Specify the different mean values for each input.
--multi_input_sample_rate “input1_name: 8000;input2_name:8000”	Specify the different sample rate values for each input.
--multi_input_down_sampling “input1_name:0;input2_name:0”	Specify the different down sampling values for each input.
--multi_input_load_address “input1_name:0;input2_name:1000”	Specify the address of the different locations to store each input.
--validation_data_path “input1_name:DATASET1_DIRECTORY; input2_name: DATASET2_DIRECTORY”	Path to directory for each input validation dataset. While creating the quantized version, this validation directory is used. A validation directory must be provided for each input in the model.

3.7. Design Restrictions

There are a few constraints and restrictions that should be kept in mind when designing a neural network with sensAI. The general hardware, software, and framework restrictions are listed below.

3.7.1. General Restrictions

The mean operation is not performed in the network itself. It must be implemented in your RTL. For more information, see the Data Preprocessing section.

To support asymmetric padding on hardware, the Convolution layer should be followed by BatchNorm operation.

3.7.2. ECP5 Restrictions

- Mean is not supported in firmware.
- Binary Convolution and Convolution: The maximum kernel size for Convolution is 9x9, while Binary Convolution has a maximum size of 3. The pad is recommended to be 1.
- If there is asymmetric padding in the convolution layer, then the convolution layer should be followed by Batch-Normalization layer.
- Pooling
  - Global Average Pooling
    - The kernel must be symmetric.
    - The stride must be 1. The pad must be 0.
  - Max Pooling
    - The kernel must be symmetric.
    - The recommended size is 2 × 2.
    - The pad must be symmetric. It is recommended to use a kernel size of 9 × 9 or smaller to reduce the number of cycles used.
- For leaky_ReLU, the negative activation slope is fixed to 1/16 in hardware. Models must be trained with alpha = 0.0625 (1/16) in leaky_ReLU.

3.7.3. ECP5 - Mobilenet Mode Restrictions

In addition to the previously-stated ECP5 restrictions, Mobilenet mode has a few additional restrictions to consider.

- Depthwise Convolution only supports kernel sizes of 3 × 3, with stride restricted to 1 or 2, and pad values restricted to 0 or 1.
- 1 × 1 convolution must have the pad set to 0.
- Mobilenet mode supports branching and merging using eltwise addition. Both inputs and outputs of eltwise addition must be in the same format [either in 16b or in 8b].
- The Depth wise kernel input is restricted to 8,192. For given channels (C, H, W), this means that (W * H/2) must be less than or equal to 8, 192.
- The number of engines cannot be changed. sensAI disables the ability to change this number to prevent generating an invalid firmware file. The number of engines used is eight convolution engines, eight depthwise convolution engines, and 64 1 × 1 convolution engines.
  - Because the eight Convolution engines are in dual core configuration, there are only four dual core engines. This is less than the limit of the normal ECP5 mode, meaning that the number of output EBRs is four when using the dual core engines instead of eight.
  - There are still eight output EBRs when using the eight depthwise convolution engines.
- ReLU6 is not supported in Neural Network Compiler 7.0. Ensure that the model does not contain this activation.
- Currently, if Mobilenet is trained with TensorFlow and the first convolution layer uses padding, the hardware simulation results may be inexact when compared to the actual hardware output. Test the hardware in this situation. The TensorFlow implementation of padding introduces differences from the present implementation employed in hardware.

3.7.4. UltraPlus Restrictions

- **Binary Convolution and Convolution**: When using a CNN design in UltraPlus, the Convolution layer should have a weight size of less than or equal to 3 and a stride (conv_stride) of 1. It is recommended to keep the pad size at 1, although larger pad sizes can be supported. Data may be lost due to the fixed-point width losing significant figures as the padding size increases. When using a BNN design on UltraPlus, the Binary Convolution layer has the same constraints as the standard Convolution layer.
  - Kernel sizes are restricted to 3 × 3 for BNN and 3 × 3 and 1 × 1 for CNN.
- **Pooling**: The Pooling layer must have a stride (pool_stride) and kernel (pool_ksize) size of 2, and a pad (pool_pad) of 0.
- Mean and Scale are not supported in firmware.
- All intermediate data in a model, except the output, is represented in unsigned 8-bit format in the hardware, using the format 1.7 to represent the data. Because of this, you should use Mean = 0 and Scale = 0.0078125 in settings for UltraPlus for any design you intend to run on the UltraPlus IP.
- Bias is not supported in the convolution layer.
- BNN supports input dimensions of 32 × 32.
- CNN supports the 32 × 32, 64 × 64, 128 × 128, and 160 × 160 input dimensions. 160 × 160 support requires Quad SPRAM.
- Unlike ECP5, there is no discrete Mobilenet mode. If a depthwise convolution is detected followed by a 1 × 1 convolution, the software will automatically generate firmware for handling Mobilenet.
- ReLU6 is not supported. Please ensure that the Mobilenet model does not contain this activation.

3.7.5. CrossLink-NX and CertusPro-NX Optimized and Extended Mode Restrictions

- CrossLink-NX and CertusPro-NX devices only support CNN designs. At this time, there is no support for BNN-based networks. Use ECP5 or UltraPlus if binary network support is required.
- Weights and activations must be quantized for CrossLink-NX and CertusPro-NX. Refer to the Fixed Point Quantization for iCE40 UltraPlus, CrossLink-NX, CertusPro-NX, and Avant section for more details on how to quantize your network correctly.
- 3 × 3 and 1 × 1 are the only supported convolution kernel sizes. The stride required to be 1 for both types (except for optimized IP, where stride 2 is also supported for 3 x 3 kernel size) . The pad can be 0 or 1 for 3 × 3 kernels, and the pad must be 0 for 1 × 1 convolution.
- Depthwise Convolution only supports 3 × 3 kernel size, with the stride required to be 1 for non-optimized IP and 1 or 2 for optimized IP. The pad can be either 0 or 1.
- Bias is not supported in the convolution layer.
- 4-bit weight quantization is only supported with the Learned Step Quantized model in Optimized IP mode.
- 2 × 2 is the only supported pooling kernel size. The stride must be 2, and the pad must be 0. Odd input to the pooling layer is not supported.
- ReLU and leaky ReLU are both supported. The negative slope for leaky ReLU must be 0.0625 (or 1/16). QuantReLU must be present after each ReLU.
- QuantReLU only supports numbits = 8, minimum = 0, and maximum = 2.
- The fully connected layer is only supported at last layer (no intermediate fully connected is supported).
- The last layer must be fully connected or CBSR. In CBSR, convolution types should be normal, depthwise, or 1 × 1 convolution.
- Mean and Scale are not supported in the firmware.
- For resize operation, bilinear and nearest-neighbor method are supported. The scale factor of 2 is only supported.
- Unlike ECP5, there is no discrete Mobilenet mode. If a depthwise convolution is detected followed by a 1 × 1 convolution, the software automatically generates firmware for handling Mobilenet.
- ReLU6 is not supported. Please ensure that the Mobilenet model does not contain this activation.

- Branching or merging structures, such as Concat and ELTwise addition, are not supported in compact mode. Use either optimized mode or extended mode if you wish to use the ELTwise or Concat operations. Also, both inputs and outputs of eltwise addition must be in 8-bit quantized format.
- CrossLink-NX and CertusPro-NX utilize external memory by allowing the base address for the data and code to be specified. As a result, you to accidentally set a start address that leaves insufficient memory available for the data or firmware. If the data base section address leaves insufficient room for the data, the analysis stage produces an error indicating this. Likewise, if the code base address leaves insufficient room for the code, the analysis stage produces an error stating as much. In either case, the address must be changed to allow for sufficient space.
- Depths/channels used in Crosslink-NX and CertusPro-NX are recommended to be multiples of 4 for depthwise and 1 × 1 convolution for better performance.
- CrossLink-NX with Quad LRAM (262,144 bytes) on-chip large memory size is available only for the CrossLink-NX 17k device, and due to the limitation of EBR on the 17k device, it will be available with a 1k scratch pad size only. Do not use firmware compiled with a Quad LRAM size for the CrossLink-NX 40k device. For CertusPro-NX, all scratch pad sizes are supported with Quad LRAM.
- Large input resolutions like VGA and QVGA are only supported in CrossLink-NX optimized mode, CrossLink-NX extended mode, CertusPro-NX optimized mode, and CertusPro-NX extended mode.
- Embedded mode is only supported for CrossLink-NX Optimized and CertusPro-NX Optimized devices.
- Embedded mode only allows dual or Quad LRAM (such as, with Embedded Mode on, you cannot use 64 KB of LRAM).
- Embedded mode does not allow external memory. If you observe a memory error, reduce the filter size or model dimension, or run the model with Embedded Mode off.
- Focus Layer is supported as the first layer only in the Optimized IP mode.
- 4-bit activation is only supported in Optimized IP mode.
- 4-bit input data to the fully connected layer is not supported.

3.7.6. CertusPro-NX and Avant Advanced CNN IP Restrictions

Currently, the CertusPro-NX and Avant devices' advanced CNN IP only support the following layers.

- Convolution (kernel size: 7 × 7, 5 × 5, 3 × 3, 1 × 1)
- Eltwise addition
- Concat
- Fully Connected
- Pooling (2 × 2 kernel, stride = 2, pad = 0)
- Pooling (K × K kernel, stride = 1, pad = K/2)
- Multiply, subtract, divide and reciprocate.
- The focus layer is currently implemented using RTL and must be part of preprocessing. It is always supported after the input layer.
- For the resize operation, bilinear and nearest-neighbor methods are supported. The scale factor of 2 is only supported.
- CertusPro-NX and Lattice Avant devices only support CNN designs. At this time, there is no support for BNN-based networks. Use ECP5 or UltraPlus if binary network support is required.
- Weights and activations must be quantized for CertusPro-NX and Lattice Avant devices. Refer to the Fixed Point Quantization for iCE40 UltraPlus, CrossLink-NX, CertusPro-NX, and Avant section for details on how to quantize your network correctly.
- 3 × 3, 1 × 1, 5 × 5, and 7 × 7 are the only supported convolution kernel sizes. The stride must be 1 for a 5 × 5 kernels; pad = 2. For 3 × 3, stride = 2 with asymmetric padding is supported to achieve output dimension (H/2, W/2). Pad = 0 is not supported with the 3 × 3 kernel.
- Depthwise Convolution supports 5 × 5, 3 × 3 kernel size, with stride = 1, and pad = 1.
- The 2 × 2 kernel is supported for pooling. Stride = 2, and pad = 0 are required. Odd input to the pooling layer is not supported.

- For pooling with a K × K kernel, stride = 1, and padding must equal K/2.
- ReLU and leaky ReLU are both supported. The negative slope for leaky ReLU must be 0.0625 (or 1/16). QuantReLU must be present before or after each ReLU.
- QuantReLU only supports numbits = 8, minimum = 0, and maximum = 2.
- The fully connected layer is supported as the last and intermediate layer. The intermediate fully connected layer must be followed by the last fully connected layer and must be quantized.
- The last layer must be fully connected, CBSR, or resized bilinear. In CBSR, convolution types should be normal, depthwise, or 1 ×1 convolution.
- Mean and Scale are not supported in the firmware.
- Unlike ECP5, there is no discrete Mobilenet mode. If a depthwise convolution is detected, followed by a 1 × 1 convolution, the software automatically generates firmware for handling Mobilenet.
- ReLU6 is not supported. Please ensure that the Mobilenet model does not contain this activation.
- CertusPro-NX and Lattice Avant devices utilize external memory by allowing the base address for the data and code to be specified. As a result, you to accidentally set a start address that leaves insufficient memory available for the data or the firmware. If the data base section address leaves insufficient room for the data, the analysis stage produces an error. Likewise, if the code base address leaves insufficient room for the code, the analysis stage produces a warning. In either case, the address must be changed to allow sufficient space for both data and code.
- Depths/channels used in CertusPro-NX and Lattice Avant are recommended to be multiples of 4 for depthwise and 1 × 1 convolution for better performance.
- Currently, 2 and 4 multiport modes are supported. These modes use more resources but speed up 1 × 1 convolution execution.
- The focus layer is supported as the first layer only in Advanced IP mode.
- 4-bit activation is not supported in Advanced IP.

3.7.7. Keras Restrictions

See the Supported Keras Layers section for more requirements for individual layers.

3.7.8. TensorFlow Restrictions

Versions 1.14, 2.0, 2.3, 2.5, and 2.9 of TensorFlow are supported by sensAI. Networks designed for other versions may not be compatible.

See the Supported TensorFlow Operations section for more requirements for individual operations.

3.7.9. AutoKeras Restrictions

- The model training was done considering a multiclass ***CLASSIFICATION*** task only.
- The model architectures were experimented with an input size of 32 × 32 × 1.
- The optimizer that AutoKeras chooses sometimes has a very small initial learning rate, and sometimes it is used along with learning rate decay, which affects training accuracy and loss. Hence, a constant optimizer was used (SGD with an initial LR=0.1 and a learning rate scheduler callback option).
- For now, the only hyperparameter that is varying is the number of channels (depth) in each layer. If the ***number of layers*** is kept as a hyperparameter, then it tries to go for a very large depth near the FC layer, and this creates the FC output value to explode. So the number of layers is now fixed.
- The ***max model size*** parameter is tested with a few experiments (with a given seed and resolution) to create a model (.bin file size) smaller than the limit for certain devices like UltraPlus.
- For reproducibility, when the seed is provided, it searches through the same hyperparameter combinations every time we run the script. However, the loss value that the AutoKeras get might differ slightly, and as a result, they may not have the same architecture as earlier. But the accuracy remains approximately within the +/-3% range.
- Note that if FC layer output crosses the range of [-32,+32], then we may experience a little higher MAE in the Neural Network Compiler, which is expected.

Refer to AutoKeras Reference Design document to know about training a model in AutoKeras for NNC.

3.7.10. ONNX Restrictions

ONNX model support is experimental. Only float and PTQ models are supported. The input to the network should be in the NCHW format. See the Supported ONNX Layers section for more requirements on individual operations.

3.8. Next Steps

Now that you have created or opened a project, you are ready to edit your project and run through the design flow, as detailed in the next section.

4. Working with Projects

4.1. Implementations

Implementations organize the structure of your design and allow you to try alternate structures and tool settings to determine which one can give you the best results. To help determine which scenario best meets your project goals, try using a different implementation of a design with different settings. Each implementation has associated active settings. When you create a new implementation, you must select its active settings.

4.1.1. Creating a New Implementation

To try a new implementation with different strategies within an existing project, you must create a new implementation.

1. Choose **File** > **Add Impl** to bring up the Implementation Options window.
2. The Implementation Options window has the same parameters as the one you encountered when creating your project initially. You can change the implementation name to a unique string if desired. Within the project, each implementation must have a unique name.
3. Change the implementation settings from the default settings, if desired.

4.1.2. Editing an Implementation

You can edit an existing implementation to change the specific input and output files, as well as the implementation settings.

1. Choose **File** > **Edit Impl** to bring up the Project Settings window.
2. The Project Implementation Settings window opens, as shown in Figure 4.1.

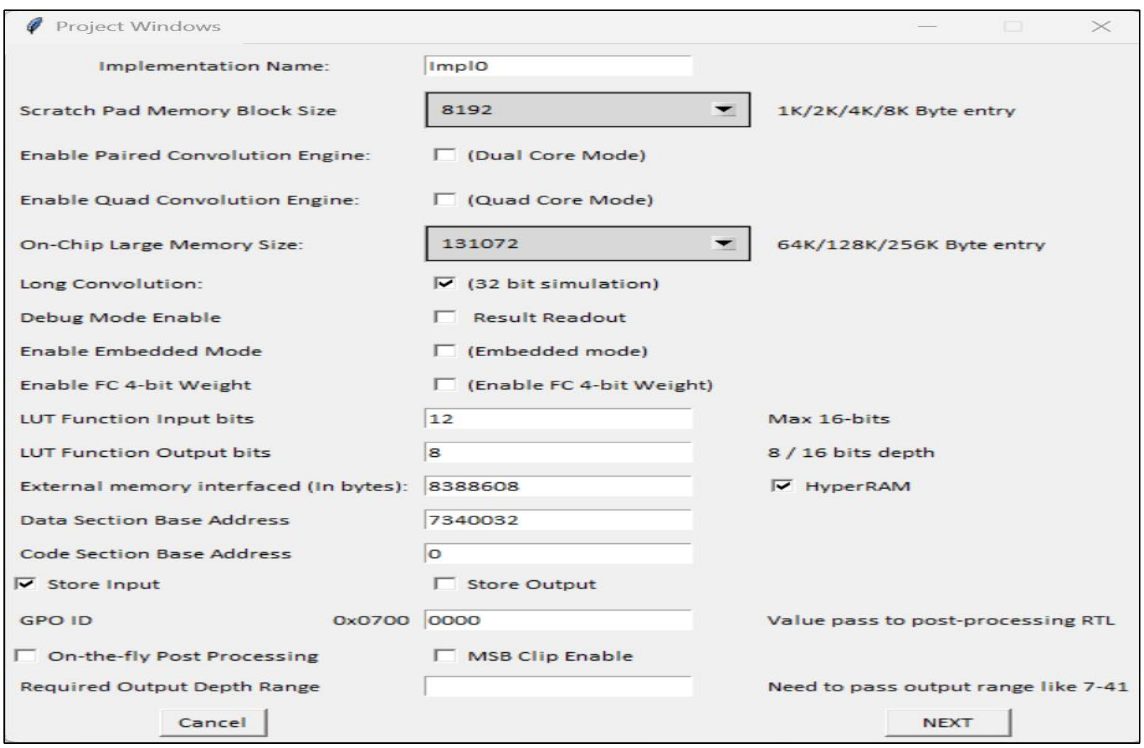


Figure 4.1. Project Implementation Options Window

3. Edit your existing settings and click **OK** to apply them to your project implementation. For more information on parameters and their limitations, refer to the Project Implementation Settings section.

4.2. Project Flow

4.2.1. Analyze

You must first run the Analyze function on your project before you can progress to the Compile or Simulate stages. It analyzes your code to verify compatibility with the Lattice CNN Compiler. You can run the Analyzer by selecting **Process > Analyze**.

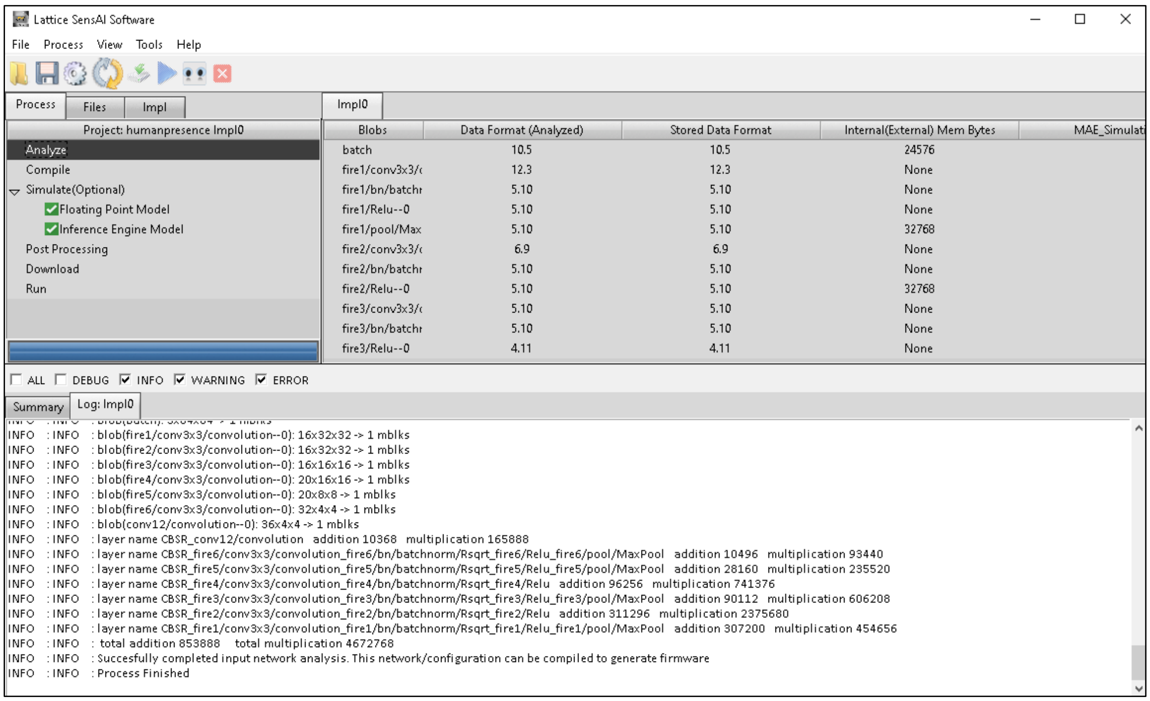


Figure 4.2. Analyze Results

After successfully analyzing a neural network file, the implementation window is updated with a set of columns listing the properties of your neural network under the current settings.

- **Blobs:** Each blob that is detected and implemented by the software is listed in this column. Some blobs that are in the network file are not implemented in the hardware, such as those used for external data processing, and are not listed here.
- **Data Format:** This column lists the breakdown of the fixed-point representation of the blob. The number preceding the period is the number of bits used to represent the integer component of the number, while the number following it is the number of bits used in the fractional component. For signed data, the total number of bits is one less than the total number of bits used, as one bit is always used for signage.
  - For clarification, the following represents a 16-bit signed number, using 15 bits to represent the integer and fraction:
    - 3.12 represents a signed number with 3 integer bits and 12 fractional bits. The sum of the two values is 15. The software thus uses a 16-bit signed format.
  - For a signed 8-bit number, the total would be 7, as shown:
    - 5.2 represents a signed number with 5 integer bits and 2 fractional bits. The sum of the two values is 7. The software thus uses an 8-bit signed format. Finally, unsigned numbers can be used in 8-bit format.

    - 5.3 represents an unsigned number. The sum of the two values is 8. The software thus uses an 8-bit unsigned format. SensAI only supports unsigned 8-bit and signed 8- and 16-bit formats. Some settings, such as layer collapse, force a certain combination of integer and fractional bits.
- **Stored Data Format**: This column is a user-editable list of the fixed-point representations of each blob. It is populated with the default values that are automatically calculated by the software. Values are written in the same format as the signed data format entry above. In order to edit the stored data format for a blob, double-click the entry in that column for the blob in question.
  You can allocate how many bits you want dedicated to the integer and fractional components for EBR storage for the specified blob. You have to specify whether the EBR accepts 16-bit mode or 8-bit mode. To use 16-bit mode, your two values need to add up to 15. To use 8-bit mode, your two values need to add up to 7.
  - 12.3 represents EBR storage in 16-bit mode with 12 integer bits and 3 fraction bits.
  - 6.1 represents EBR storage in 8-bit mode with 6 integer bits and 1 fraction bit.
- **Required Memory Bytes:** The memory required to implement each blob is listed in this column. See the Project Implementation Settings section for more details on the effects your settings may have on this.
  - **UltraPlus:** Lists the required SPRAM.
  - **CrossLink-NX, CertusPro-NX, and ECP5**: Lists the required internal (LRAM/EBR), and external (HRAM/DRAM) memory.
- **Distribution of Input Data into Memory Blocks**

  During the analysis process, input data is divided into memory blocks based on the input layer dimension. The following subsections explain the details of how this division is handled. This example uses a three-channel BGR input, though your data input may use more or less than three channels.
  - Fraction setting of the input layer: If the input values can fit in 8 bits, then the fraction settings to store input data are in 8-bit (byte mode). Hence, 16384 (for ECP5) input values can fit in a single memory block; otherwise, 8192 values can be stored in one memory block.
  - Based on the values that can fit into a single memory block (16384 total values for byte mode on ECP5), there could be four different conditions: cases where all the channels fit into a single memory block, cases where at least one channel can fit into a single memory block, cases where a single channel cannot fit into a memory block, and cases when memory blocks are not sufficient to fit input data.
    - All the channels (BGR) can fit in a single memory block.
      If the input dimensions are 3 × 32 × 32, then the total number of input values is 3,072, which is less than 16,384 values.
      In this case, all the data values are stored in a single memory block in sequential order. In this example, input data is stored in the first memory block, from address 0 to address 3071.
    - At least one channel can fit in a single memory block:
      There is also the case where all of the channels cannot fit into a single memory block, but it is still possible to put one or more channels into one.
      For cases where only a single channel can fit within a memory block, consider a case where the input dimension is 3 × 128 × 128. This corresponds to 49,152 entries, which cannot fit into a single memory block. However, a single channel has a size of 1 × 128 × 128. This is 16,384 values, which can fit within a single memory block.
      In this case, data is divided into 3 memory blocks, and each memory block can have a single channel of data values.
      Note: Even if there is some extra space remaining in the memory block, the next channel values are not stored in that space unless a second channel can fit within, as explained in the next subsection.
      In another example, consider an input dimension of 3 × 90 × 90. Once again, all three channels correspond to a size (24,300), which cannot fit within a single memory block. Even though two channels would take up 2 × 90 × 90, or 16,200 entries, which can fit in a single memory block, data is divided into memory blocks equally.

In this case, the data is divided into three memory blocks. The first memory block has the data from the first (B), the second memory block has the second (G) channel, and the third memory block has the data from the third (R) channel.
In this case, the last 8,284 values of each memory block are not used.

- A single channel cannot fit in a single memory block, but memory blocks are sufficient to fit input data.
Consider a larger network with input dimensions of 3 × 224 × 224. In this case, there are 150,528 input values, which is far too large for a single memory block. Additionally, a single channel (1 × 224 × 224) has 50,176 values, which is still too large for a single memory block.
Because of this large size, the Analyze stage attempts to divide each single channel into smaller pieces that can fit in each memory block using the following three steps:
    1. Calculate the required memory per depth:
    Number of memory blocks = Ceiling [(224x224)/16,384] = 4
    In this case, the memory per depth is 4.
    2. Calculate the height per memory block:
    Height per memory block = Total height / memory per depth value
    For a total height of 224 divided by a depth of 4, this results in a height per memory block of 224/4, which is 56 in one memory block.
    3. Because there are 4 memory blocks per depth and 3 channels, a total of 12 memory blocks are used to store the input data.

    Because each memory block stores the values of 56 heights (56 x 224), it uses 12,544 entries per memory block, and the remaining space in each memory block is unused. In this case, the data is divided as listed below:
    - 1st memory block: Channel 0 (B) 0 – 55 height values
    - 2nd memory block: Channel 0 (B) 56 – 111 height values
    - 3rd memory block: Channel 0 (B) 112 – 167 height values
    - 4th memory block: Channel 0 (B) 168 – 223 height values
    - 5th memory block: Channel 1 (G) 0 – 55 height values

    .
    .
    .
    - 11th memory block: Channel 2 (R) 112 – 167 height values
    - 12th memory block: Channel 2 (R) 168 – 223 height values

- Memory blocks are not sufficient to fit input data.
Consider a larger network with input dimensions of 3 × 300 × 300. In this case, there are 270,000 input values, which is too large for all memory blocks, where the total memory size of all blocks is 162,144 (16 × 16384). In cases where the total memory block size is not enough to store all input channels, DRAM is required to store input data. For the input layer, you need to enable the ***Store Input*** option. For intermediate layers, the DRAM address is auto assigned. During processing, data is copied from DRAM to EBR. Because of this large size, the Analyze stage attempts to divide each single channel into smaller pieces that can fit in one memory block, as above. Analyze flow assigns one or more memory blocks to process data in the engine. As data is already in DRAM, the same memory block(s) can be reused for the next piece. So even if data cannot fit into assigned memory blocks, it is not overwritten. In this case, the data is divided as listed below:
    - 1st memory block: Channel 0 (B) 0 – 50 height values
    - 1st memory block: Channel 0 (B) 51 – 100 height values
    - …
    - 1st memory block: Channel 0 (B) 251 – 300 height values
    - 2nd memory block: Channel 1 (G) 0 – 50 height values
    - 2nd memory block: Channel 1 (G) 51 – 100 height values
    - …

- 2nd memory block: Channel 1 (G) 251 – 300 height values
- 3rd memory block: Channel 2 (R) 0 – 50 height values
- 3rd memory block: Channel 2 (R) 51 – 100 height values
- …
- 3rd memory block: Channel 2 (R) 251 – 300 height values

4.2.2. Analyzer for USB Debugging

To debug ECP5 via the USB interface, this checkbox should be enabled. The analyzer adds the required external memory address information to the output files.

For ECP5, layer outputs are read out after running. As a result, the outputs of layers that have their outputs overwritten by subsequent layers cannot be read directly.

4.2.3. Compile

You can create a firmware file for your analyzed network by running the compilation flow. This generates an lscml-type file, which can be used to download the network to your hardware by the software or by another tool. You can run the compiler by selecting **Process > Compile**.

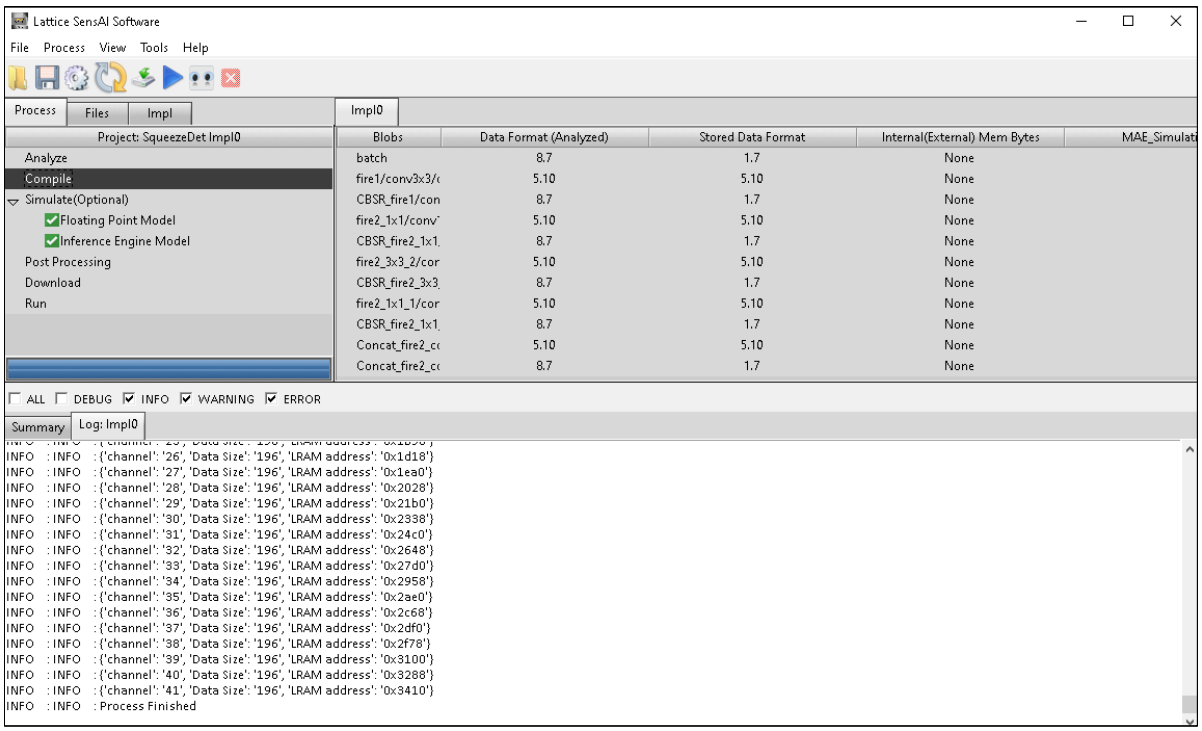


Figure 4.3. Compile Results

After your network has been successfully compiled, you are presented with performance information. ECP5 designs also report details on channel/height storage and the start/end addresses for each input EBR. The cycles used by your neural network given the specified settings are reported, with a breakdown of cycles spent on DRAM access, convolution, pooling, fully connected, and scale.

- DRAM: These are the cycles that are spent accessing or storing data in the DRAM. Designs that use more of the EBR for storage will have fewer cycles used in the DRAM stage, and this number will increase as your settings offload more storage from the EBR to the DRAM.

- Conv: The cycles used in performing convolution are reported here. In a conventional neural network, this represents the standard convolution cycle. In a binary neural network, it displays the cycles used during binary convolution. In designs utilizing EBR, it typically represents the largest share of cycles in your design.
- Pool: These cycles are used to implement pooling in your neural network.
- FC: This entry corresponds to cycles used to implement fully connected (or inner product) vector operations.
- Scale: Scaling cycles are spent performing the scale operation.

4.2.4. Simulate

It is recommended that you run the simulation to verify the results. This is not a required step to compile your project. You can simulate your analyzed network using the Simulate feature. By selecting the green or red check boxes in the process window of the left pane, the simulation type can be changed between the floating-point network, fixed-point network, or inference engine model. By default, all types of simulation are selected. You can run the simulator by selecting **Process > Simulate**.

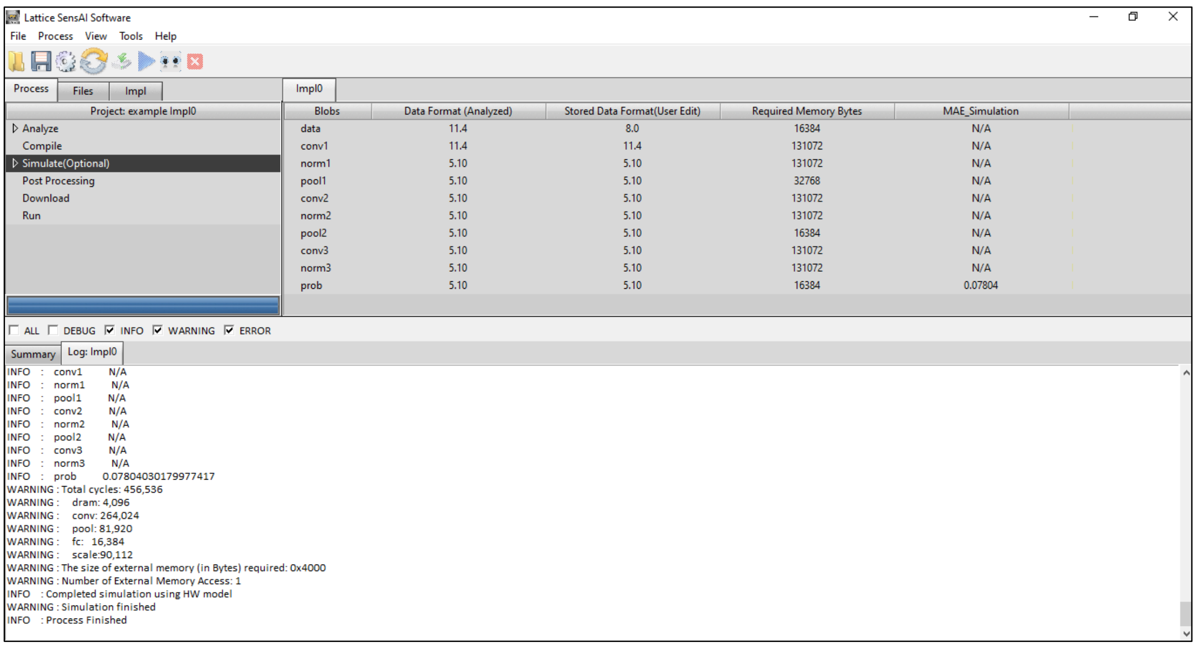


Figure 4.4. Simulate Results

The inputs and outputs of the simulation are determined by your neural network and your source file. The total cycles reported are identical to those found in the compilation stage.

Data Histogram Graph

After the analysis is complete, you can double-click on the blob name in the implementation window to view the data histogram for the particular blob.

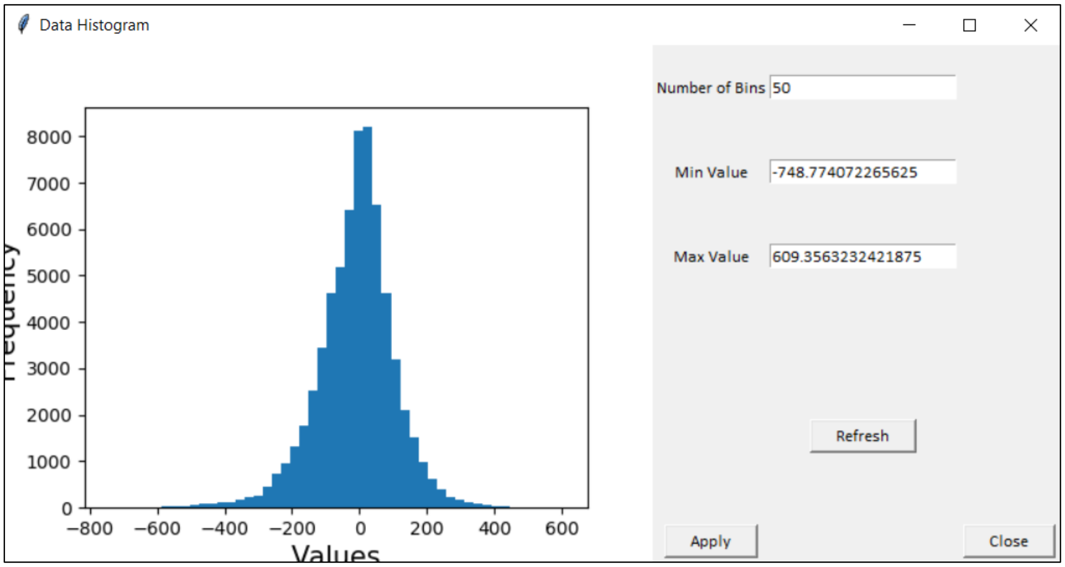


Figure 4.5. Data Histogram for the Blob

A data histogram provides information on the minimum and maximum values and distribution of data. The histogram also helps to derive the proper fraction for the blob. Clicking on **Apply** can select a frac value, so it can store the maximum (on both positive and negative) possible values.

Note: The data histogram is only available for ECP5 and UltraPlus devices.

4.2.5. Post Processing

If the Post Processing command is configured in the project setting as shown in Figure 3.2, this operation runs the post processing script on the input data (a selected image or ***.npy***) with the simulation result ***.npy*** file. You can run post processing by selecting **Process > Post Processing** as shown in Figure 4.6.

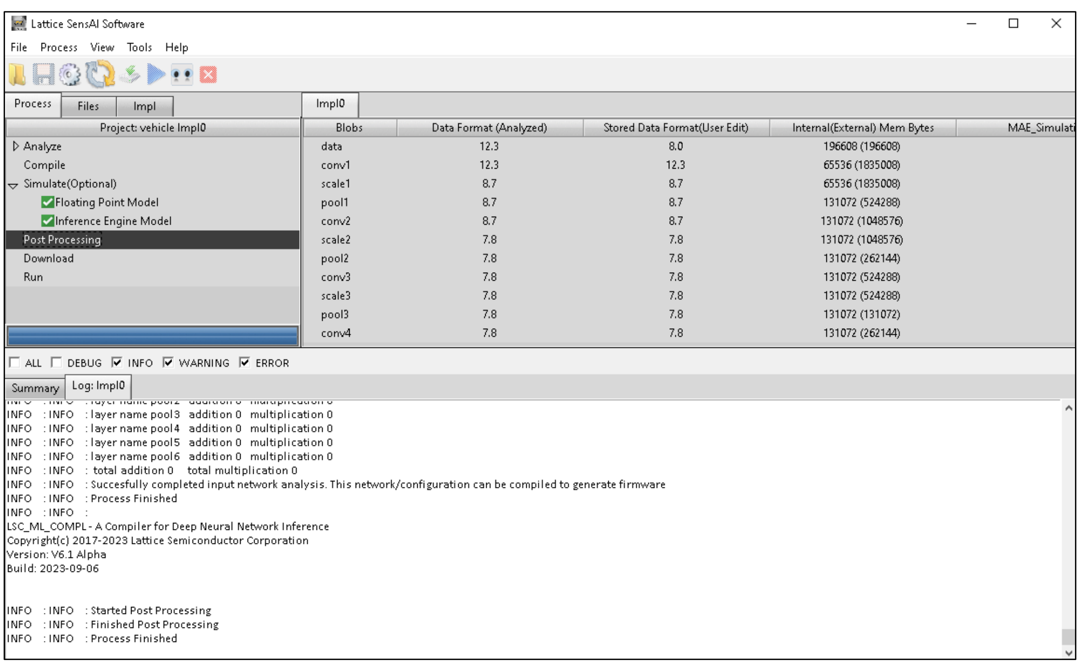


Figure 4.6. Post Processing

4.2.6. Download

Lattice Neural Network Compiler Software is capable of directly downloading a project to a compatible board that is connected to the computer. The test board must be connected via USB. You can run the download tool by selecting **Process > Download**. See the SB Debugging section for more information on the USB debugger.

4.3. Views

The **View** menu in the software allows you to view the input network, analyzed network, log file, and simulation data graph in different windows. Also, it allows users to select GUI themes.

4.3.1. Input Network

The Input Network view displays a visualization of your input network, consisting of the layers, blobs, and connections in your network file.

TensorFlow-Keras Input Network

This option opens the TensorBoard graph in your default browser, as seen in Figure 4.7.

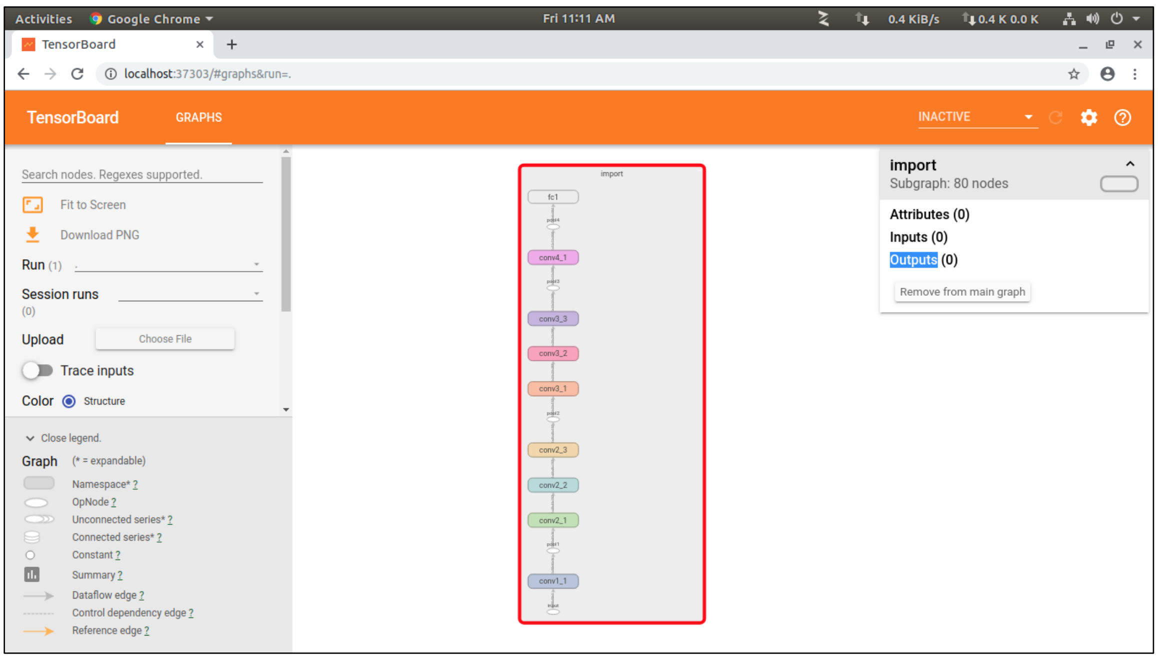


Figure 4.7. Input Network – TensorFlow or Keras

Close Tensorboard

When you return to the sensAI tool, you are asked if you wish to close the Tensorboard process. If you choose not to close, you can close it later from upper left corner tool bar as shown in Figure 4.8.

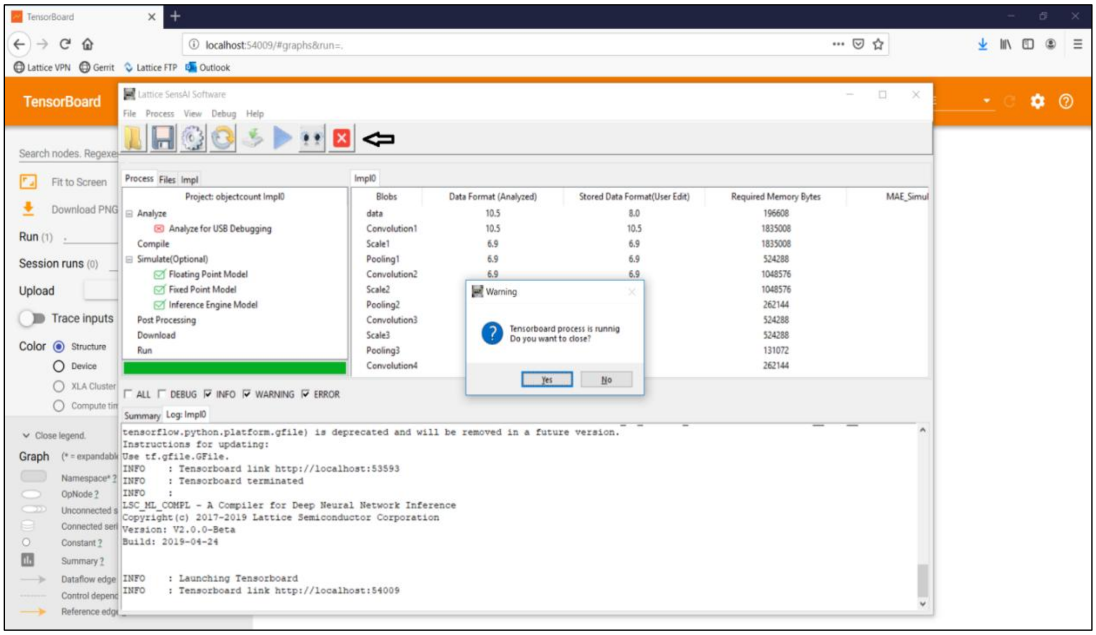


Figure 4.8. Close Tensorboard Process

4.3.2. Analyzed Network

The Analyzed Network view displays a visualization of your analyzed network. This is only available after the analyze stage of the project flow. In addition to its entry in the view menu, you can also click the **View Analyzed Network** button to the right of the **Run** button to bring up the display.

4.3.3. GUI Themes

The GUI Themes menu (Figure 4.9) allows you to update the look of sensAI. Simply click on one of the many options to choose the theme that suits you.

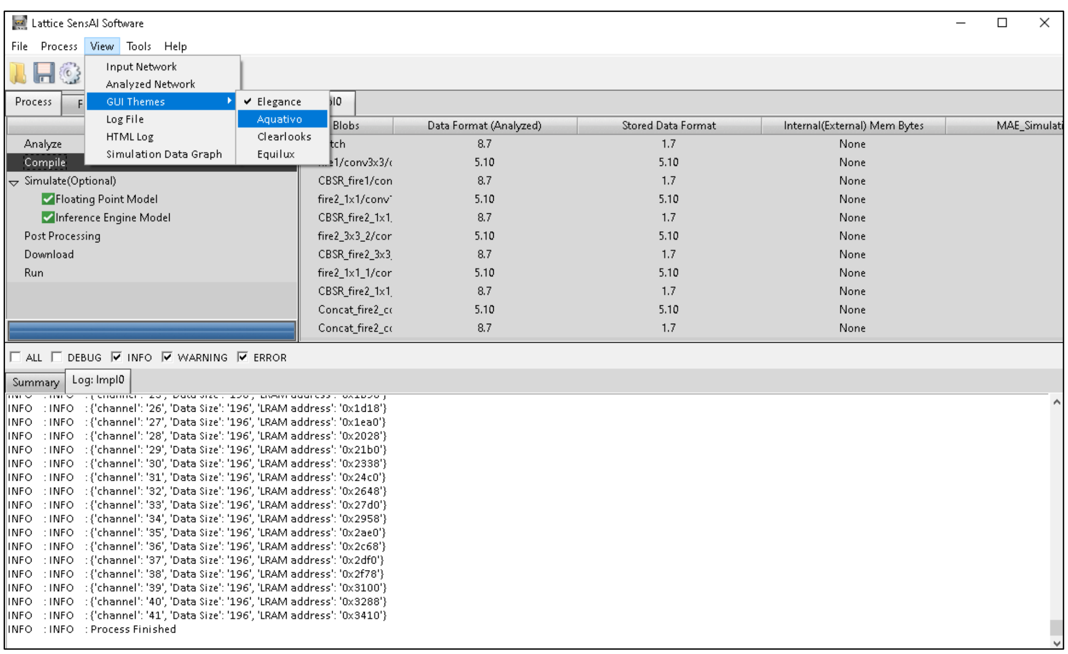


Figure 4.9. GUI Themes

4.3.4. Log File

The Log File view allows you to view the output log of your project. This is a history of operations you have initiated and the output that was generated as a result. If you would prefer to use a text viewer of your choice, the contents of your log file are stored in a ***.log*** file in your project directory.

4.3.5. HTML Log File

This HTML log file is simply a view of log files in HTML pages. You can open the HTML log in two ways. You can open an HTML log webpage by clicking **View > HTML log**, as shown in Figure 4.10. When you open the same project multiple times, new HTML pages are created. When you open the HTML log in your browser, there are four log sections: debug, info, warning, and error. There are refutations of each section's arguments. The default view of this webpage is a combination of four sections. Whenever you click on any section, they show the log of each section donly. There is a search option available for each section. Figure 4.11 shows the default view of the HTML log. Figure 4.12 shows the search option for the warning. The background colors for each portion are different.

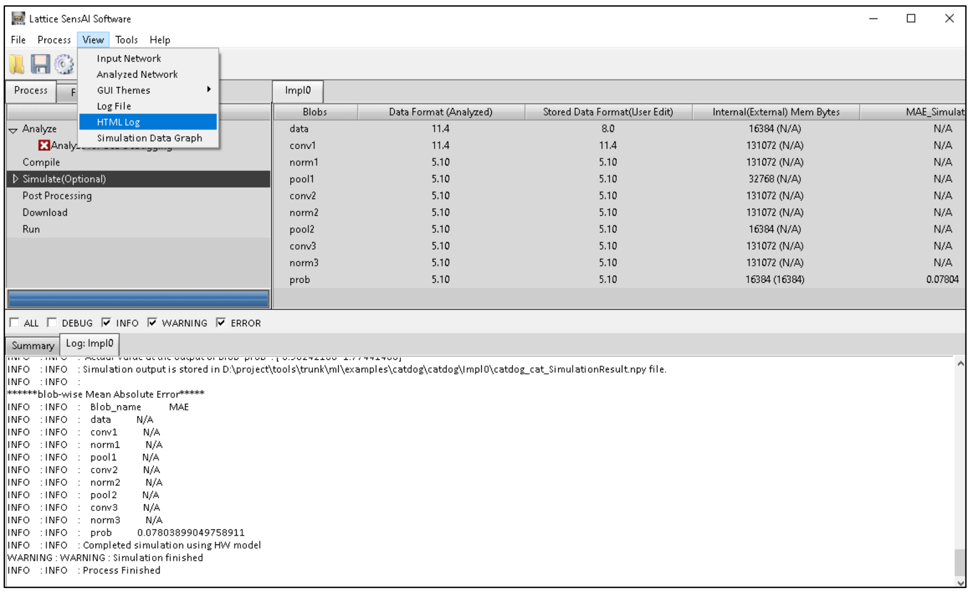


Figure 4.10. HTML Log

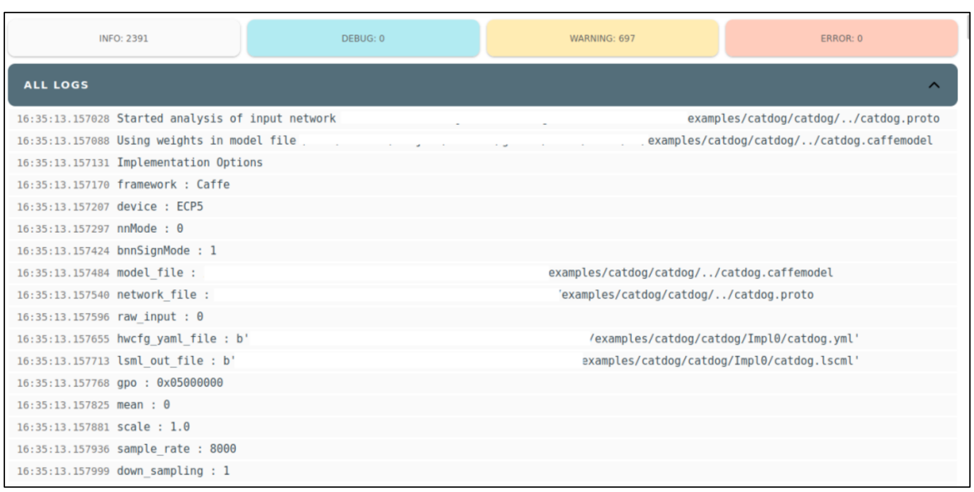


Figure 4.11. Default View of HTML log

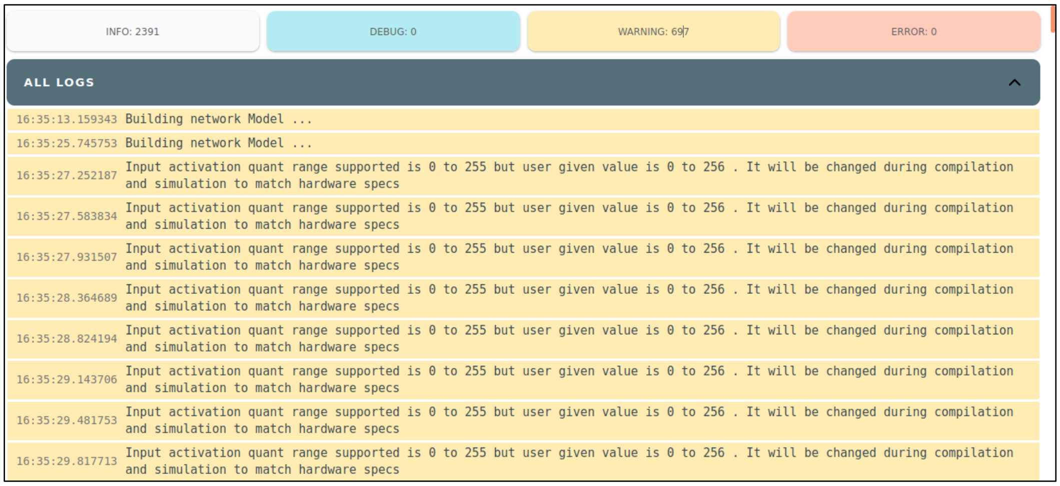


Figure 4.12. Search Functionality of Warning

4.3.6. Simulation Data Graph

The simulation data graph (Figure 4.13) shows the comparison of the predicted values of the floating-point network, fixed-point network, and hardware after running the simulation step. This view is accessible after completing a software simulation. The graph can zoom in or out, and it allows you to configure subplots and export them as an image or a PDF file.

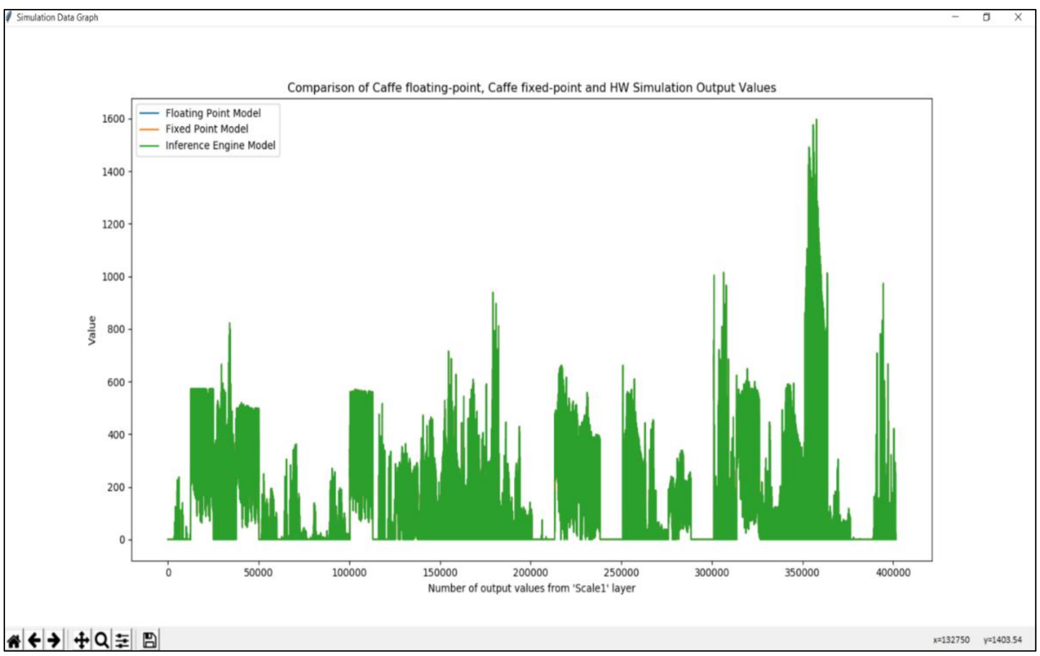


Figure 4.13. Simulation Data Graph

4.4. Example Projects

This section provides project samples that you can work on to become more familiar with the software before starting your own project.

The Neural Network Compiler includes several example projects as a reference for using the tool. The CatDog and HumanPresence projects can be loaded from the sensAI user interface and run through the analysis, compilation, and simulation stages. The post processing, meanwhile, contains a Yolo vehicle detection post processing operation script for the given input image and last layer output data (.npy).

4.4.1. Humanpresence

The *humanpresence* example network can take an input image of size 64 × 64 × 3 and determine humans in it. The accuracy depends on the images it was trained with and the test image used.

To launch the *humanpresence* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
1. Click on **File > Open**. You can also click the **Open File** button.
2. Navigate to the *examples/humanpresence* directory and select *humanpresence.ldnn*. Click **Open**.

   This loads the *humanpresence* project.

Now that the project is loaded, you are able to use several features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. To do this, click **Process > Compile** from the menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network with a single click.
3. After the network is compiled and analyzed, run the simulate function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the user interface to enable or disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. The **Floating-Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can see the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to enable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.2. GoogleNet

This *GoogleNet* network example can take an input image of size 224 × 224 × 1 and determine the number of humans in the image. The accuracy depends on the images it was trained with and the test image used.

To launch this *GoogleNet* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
2. Click **File > Open**. You can also click the **Open File** button.

   Navigate to the *examples/GoogleNet* directory and select *GoogleNet.ldnn*. Click **Open**. This loads the *GoogleNet* project.

Now that the project is loaded, you are able to use several of the features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. You can click **Process > Compile** from the menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button in the GUI. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network with a single click.
3. After the network is compiled and analyzed, run the simulation function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the GUI in order to enable and disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. The **Floating-Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can view the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to re-enable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.3. SqueezeDet

This *SqueezeDet* example network can take an input image of size 224 × 224 × 1 and determine the number of humans in the image. The accuracy depends on the images it was trained with and the test image used.

To launch this *SqueezeDet* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
2. Click on **File > Open**. You can also click the **Open File** button.

   Navigate to the *examples/SqueezeDet* directory and select *SqueezeDet.ldnn*. Click **Open**. This loads the *SqueezeDet* project.

Now that the project is loaded, you are able to use several features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. To do this, click **Process > Compile** from the dropdown menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network with a single click.

3. After the network is compiled and analyzed, run the simulate function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the user interface to enable or disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. The **Floating-Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can see the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to re-enable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.4. Handgesture

This *Handgesture* example network can take an input image of size 32 × 32 × 1 and determine hand gesture in the image. The accuracy depends on the images it was trained with and the test image used. The *Handgesture* model is non-quantized. Lattice has quantized it using the Post Training Quantization Flow of SensAI.

To launch this *Handgesture* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
2. Click on **File > Open**. You can also click the **Open File** button.

   Navigate to the *examples/Handgesture* directory and select *Handgesture.ldnn*. Click **Open**. This loads the *Handgesture* project.

Now that the project is loaded, you are able to use several features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. To do this, click **Process > Compile** from the dropdown menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network in a single click.
3. After the network is compiled and analyzed, run the simulate function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the user interface to enable or disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. **Floating Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can see the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, or any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to reenable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.5. MV1 (MobileNet V1)

This *MV1* example network can take an input image of size 240 × 320 × 1 and detect barcode in the image. The accuracy depends on the images it was trained with and the test image used.

To launch this *MV1* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
2. Click on **File > Open**. You can also click the **Open File** button.
3. Navigate to the *examples/MV1* directory and select *MobileNet_v1.ldnn*. Click **Open**.

   This loads the *MobileNet v1* project.

Now that the project is loaded, you are able to use several features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. To do this, click **Process > Compile** from the dropdown menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network in a single click.
3. After the network is compiled and analyzed, run the simulate function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the user interface to enable or disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. **Floating Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can see the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, or any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to reenable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.6. MV2 (MobileNet V2)

This *MV2* example network can take an input image of size 240 × 320 × 1 and detect barcode in the image. The accuracy depends on the images it was trained with and the test image used. This model is trained with the Learned Step Quantization (LSQ) technique.

To launch this *MV2* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
2. Click on **File > Open**. You can also click the **Open File** button.
3. Navigate to the *examples/MV2* directory and select *MobileNet_V2.ldnn*. Click **Open**.

   This loads the *MobileNet v2* project.

Now that the project is loaded, you are able to use several features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. To do this, click **Process > Compile** from the dropdown menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network in a single click.

3. After the network is compiled and analyzed, run the simulate function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the user interface to enable or disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. **Floating Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can see the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, or any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to reenable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.7. YoloV5

This *YoloV5* example network can take an input image of size 160 × 160 × 1 and detect barcode in the image. The accuracy depends on the images it was trained with and the test image used.

To launch this *YoloV5* project:

1. Launch the Lattice sensAI Neural Network Compiler software.
2. Click on **File > Open**. You can also click the **Open File** button.
3. Navigate to the *examples/YoloV5* directory and select *YoloV5.ldnn*. Click **Open**.

   This loads the *YoloV5* project.

Now that the project is loaded, you are able to use several features of the software.

To analyze, compile, and simulate the project:

1. Click **Process > Analyze** from the menu.
2. After the network is analyzed, compile the project. To do this, click **Process > Compile** from the dropdown menu.

   Note: You can combine these steps by clicking the **Analyze and Compile** button. The Lattice sensAI Neural Network Compiler software analyzes and then compiles the network in a single click.
3. After the network is compiled and analyzed, run the simulate function. Click on the checkmarks under the **Simulate (Optional)** category on the left-hand side of the user interface to enable or disable different types of simulation. Click on the checkmarks to the left of **Fixed-Point Model** and **Inference Model** to disable them. **Floating Point Model** is the only option with a green checkmark at this point.
4. Click **Project > Simulate** from the menu.
5. When the process is completed, you can see the floating-point model output.

You can try running other simulation types. You can run one simulation at a time, or any two, or all three simulation types at one time. For this example, click on the x marks to the left of **Fixed Point** and **Inference Model** to reenable them. Click **Project > Simulate** again. Your output now includes the results of all three models, rather than just the floating-point model.

4.4.8. YoloV8

This example network can take an input image of size 96 × 128 × 1. The accuracy depends on the dataset it was trained with and the test image used. You can find the project file under the *examples/ Facedetection_Yolov8* directory. The steps to load and run the model are the same as *YoloV5*. Refer to the *YoloV5* example for more details.

4.4.9. YoloV11

This example network can take an input image of size 1 x 288 x 384 x 3. The accuracy depends on the dataset it was trained with and the test image used. You can find the project file under the *examples/ Yolov11* directory. The steps to load and run the model are the same as *YoloV5*. Refer to the *YoloV5* example for more details.

4.4.10. Resenet18v1

This example network can take an input image of size 1 x 288 x 384 x 3. The accuracy depends on the dataset it was trained with and the test image used. You can find the project file under the *examples/ Resnet18* directory. The steps to load and run the model are the same as *YoloV5*. Refer to the *YoloV5* example for more details.

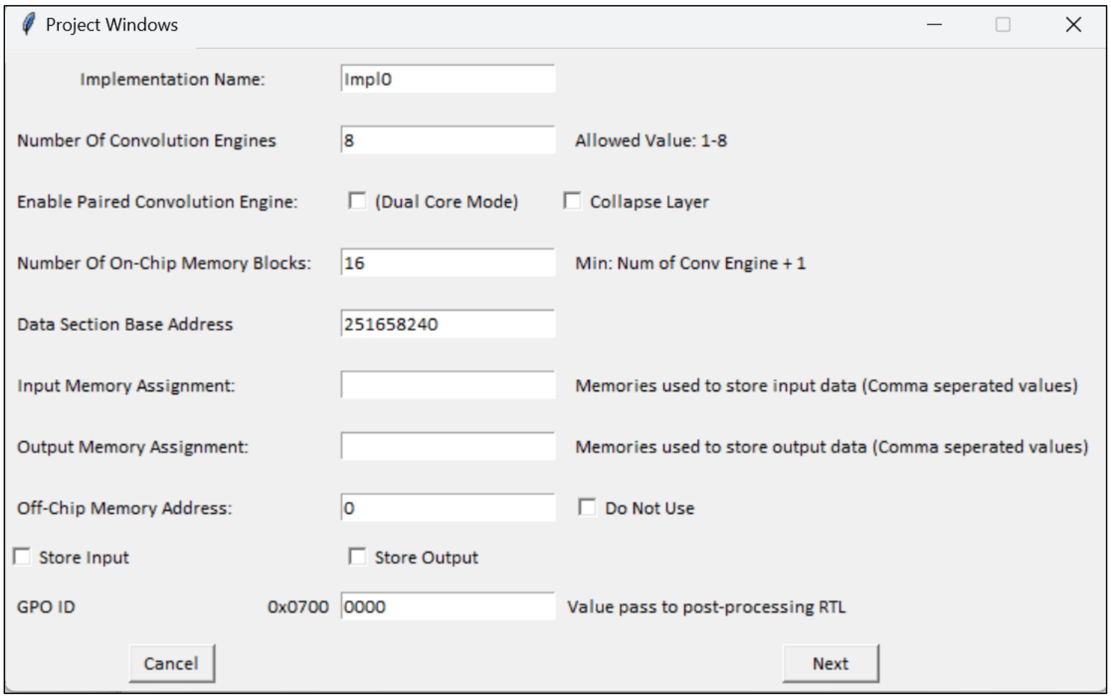
4.4.11. Toy_mnist

This example network can take an input image of size 28 × 28 × 1 and recognize digit. The accuracy depends on the data set it was trained with and the test image used. You can find the project file under the *examples/toy_mnist* directory. The steps to load and run the model are the same as *YoloV5*. Refer to the *YoloV5* example for more details.

5. Advanced Topics

5.1. Project Implementation Settings

Each project has several main settings for customizing your neural network implementation. These settings are accessed either during new project creation (see the Creating a New Project section) or by editing an existing implementation (see the Editing an Implementation section). These settings are visible in the Project Implementation Window, as shown in Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8, Figure 5.9, and Figure 5.11. We have only shown Post processing screenshot for ECP5 but it will be common for all IP's.



On clicking next, you will see window for Parameter selection for post processing

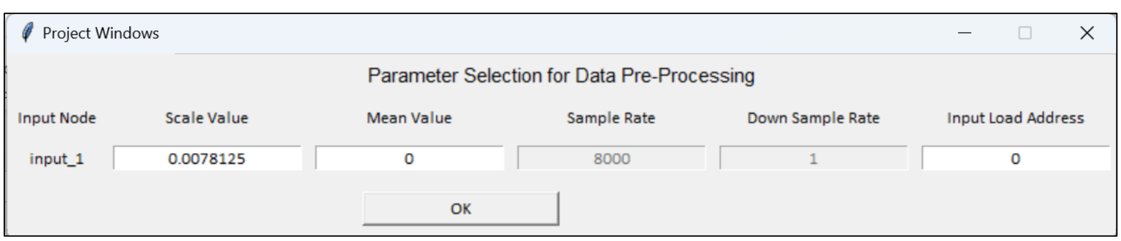


Figure 5.1. Project Implementation Window – ECP5

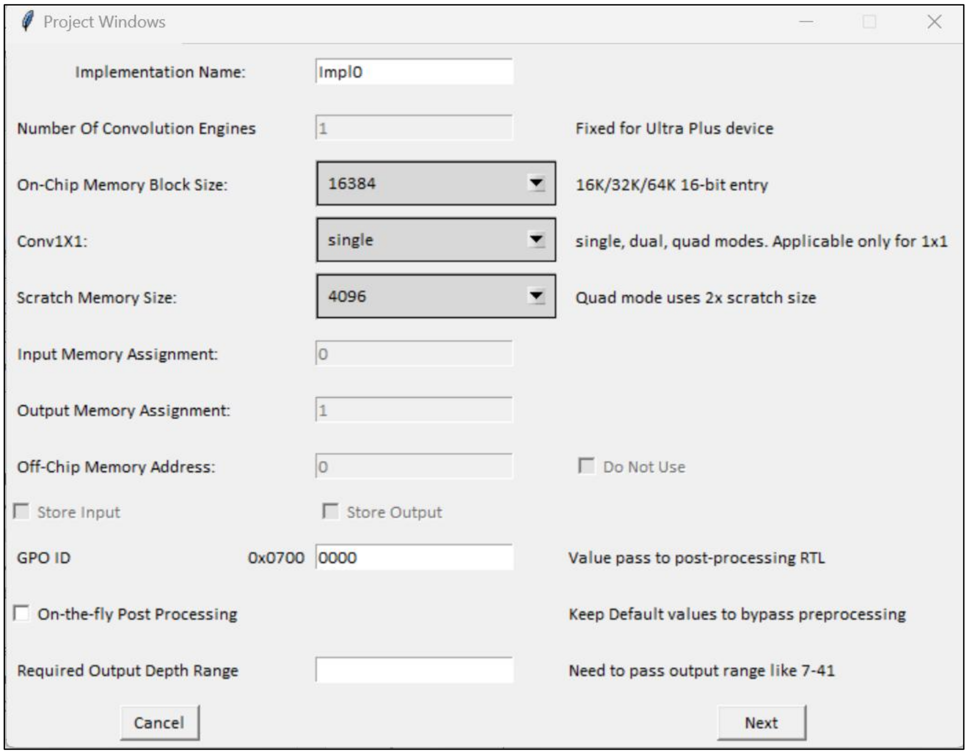


Figure 5.2. Project Implementation Window – UltraPlus (1)

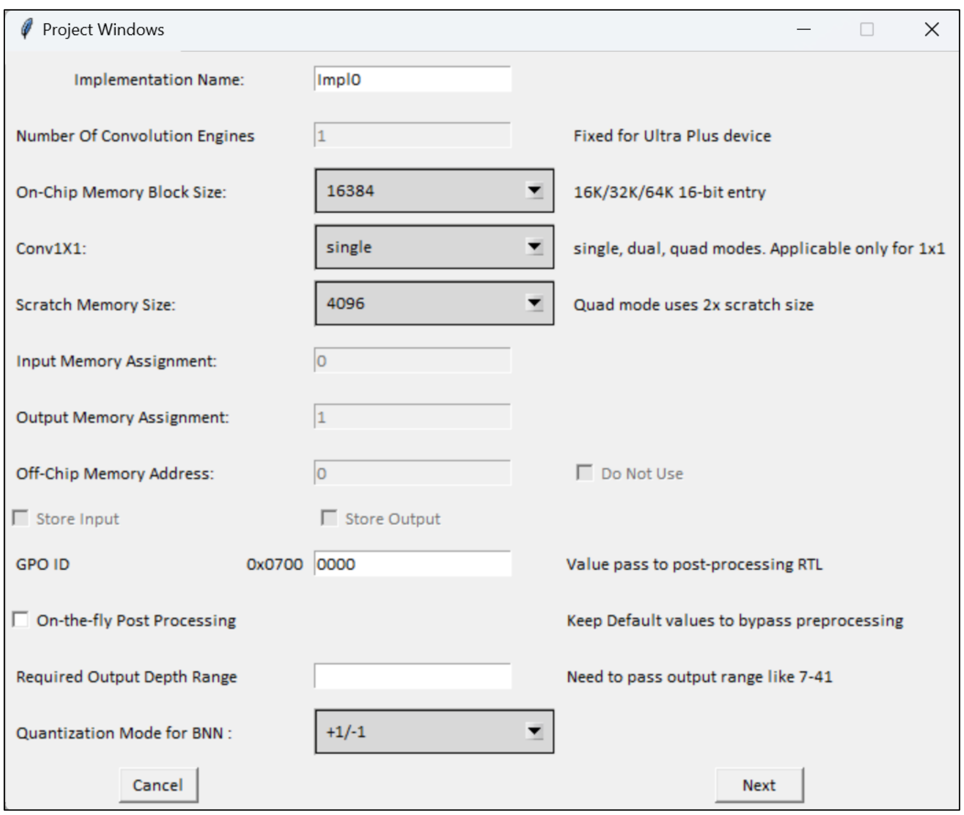


Figure 5.3. Project Implementation Window – UltraPlus (2)

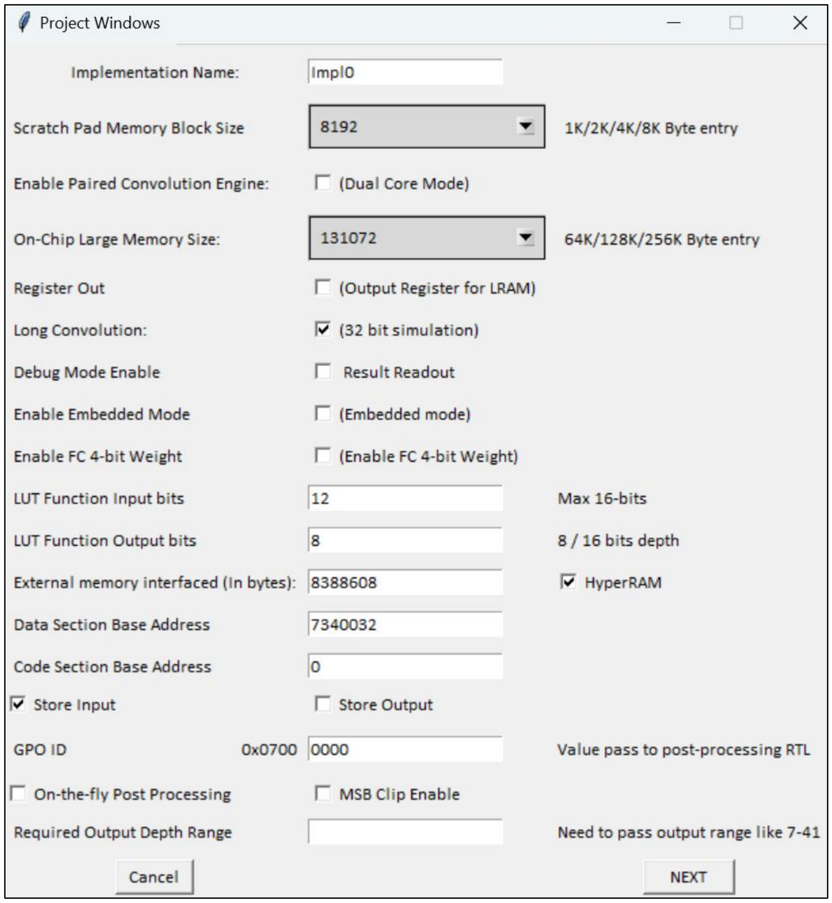


Figure 5.4. Project Implementation Window – CrossLink-NX-Optimized

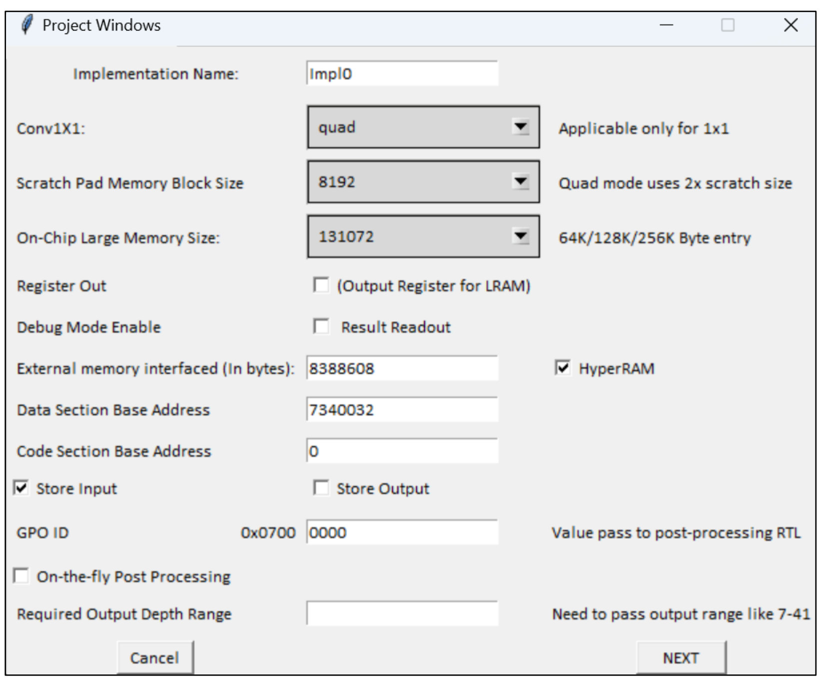


Figure 5.5. Project Implementation Window – CrossLink-NX-Compact

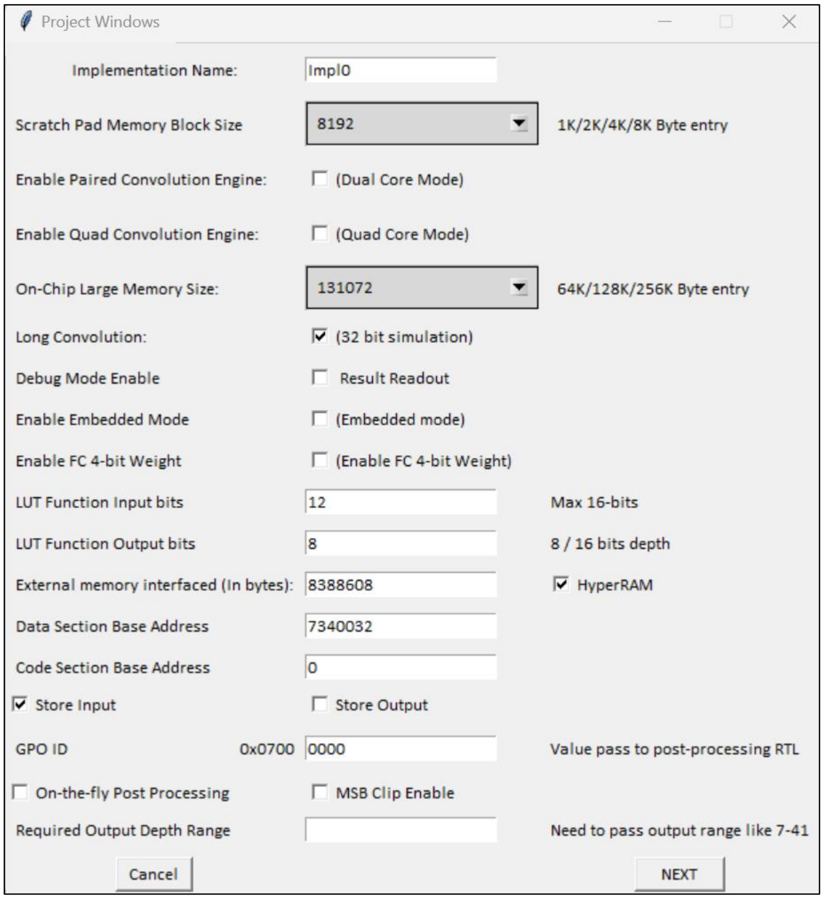


Figure 5.6. Project Implementation Window – CertusPro-NX-Optimized

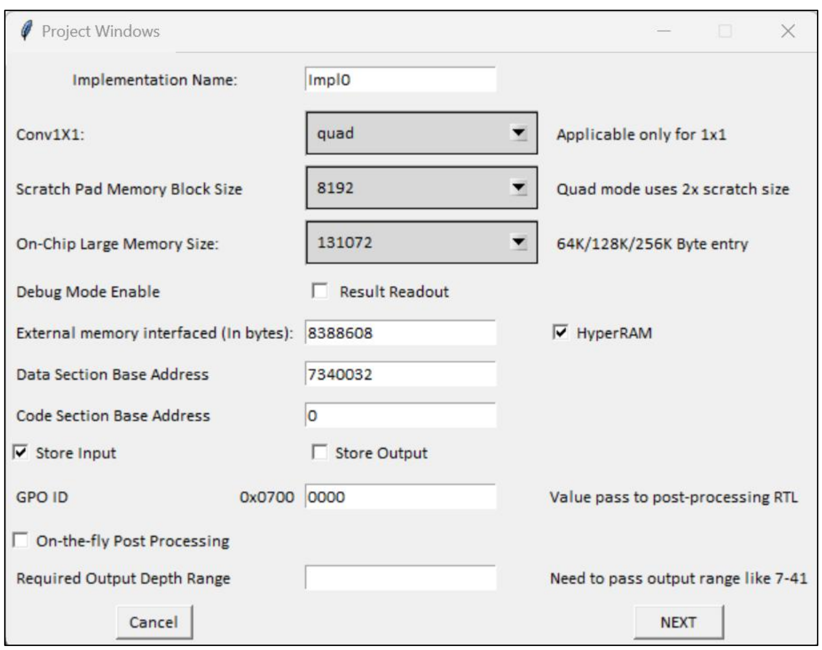


Figure 5.7. Project Implementation Window – CertusPro-NX-Compact

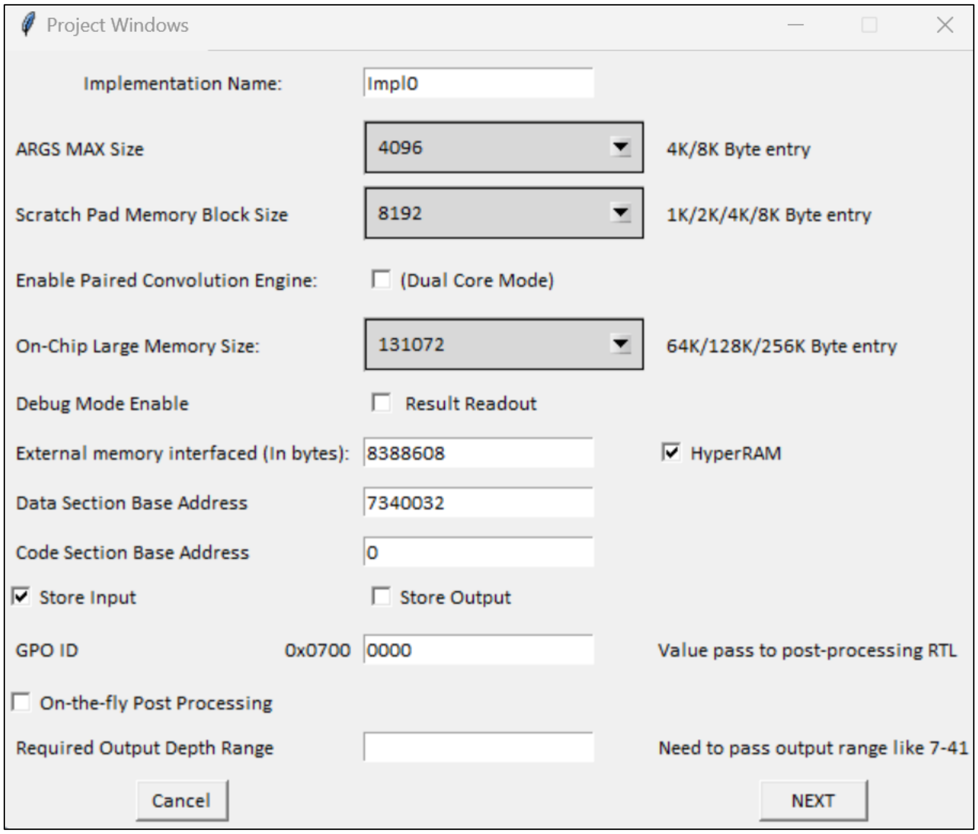


Figure 5.8. Project Implementation Window – CertusPro-NX-Extended

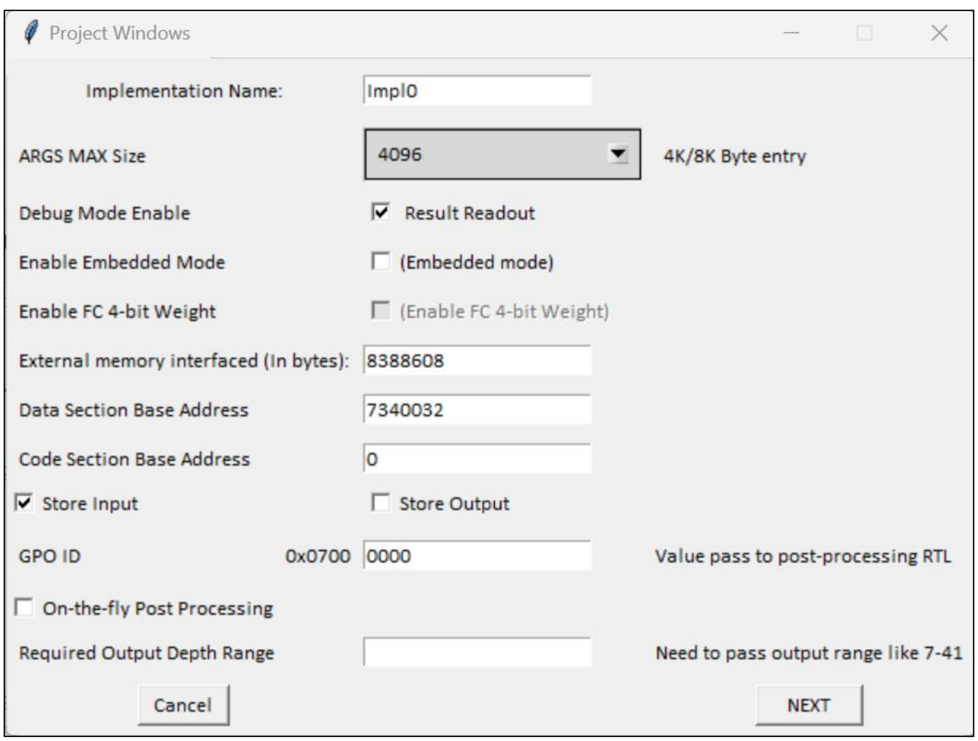


Figure 5.9 Project Implementation Window – CertusPro-NX Advanced IP Part 1

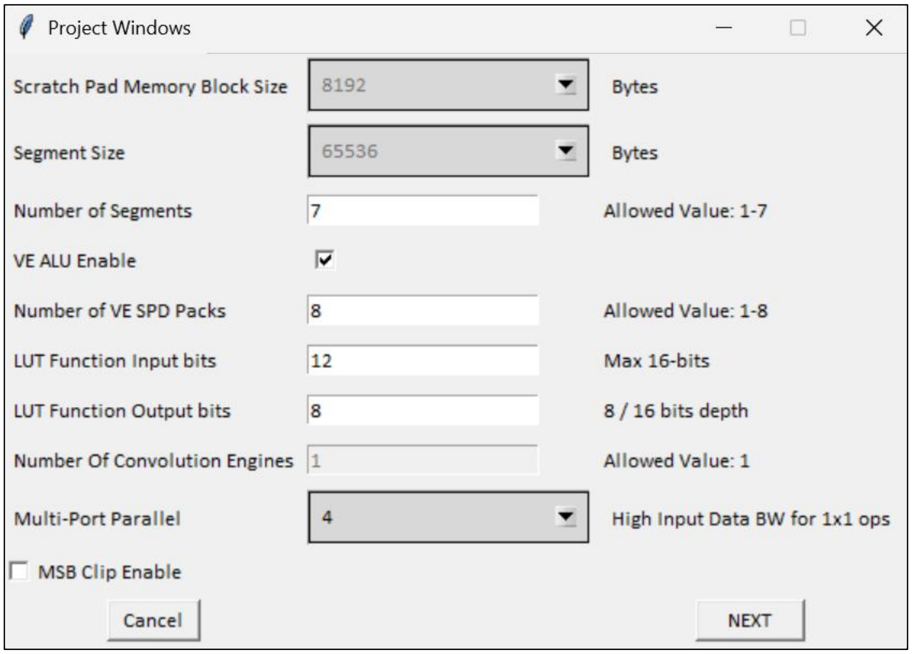


Figure 5.10 Project Implementation Window – CertusPro-NX Advanced IP Part 2

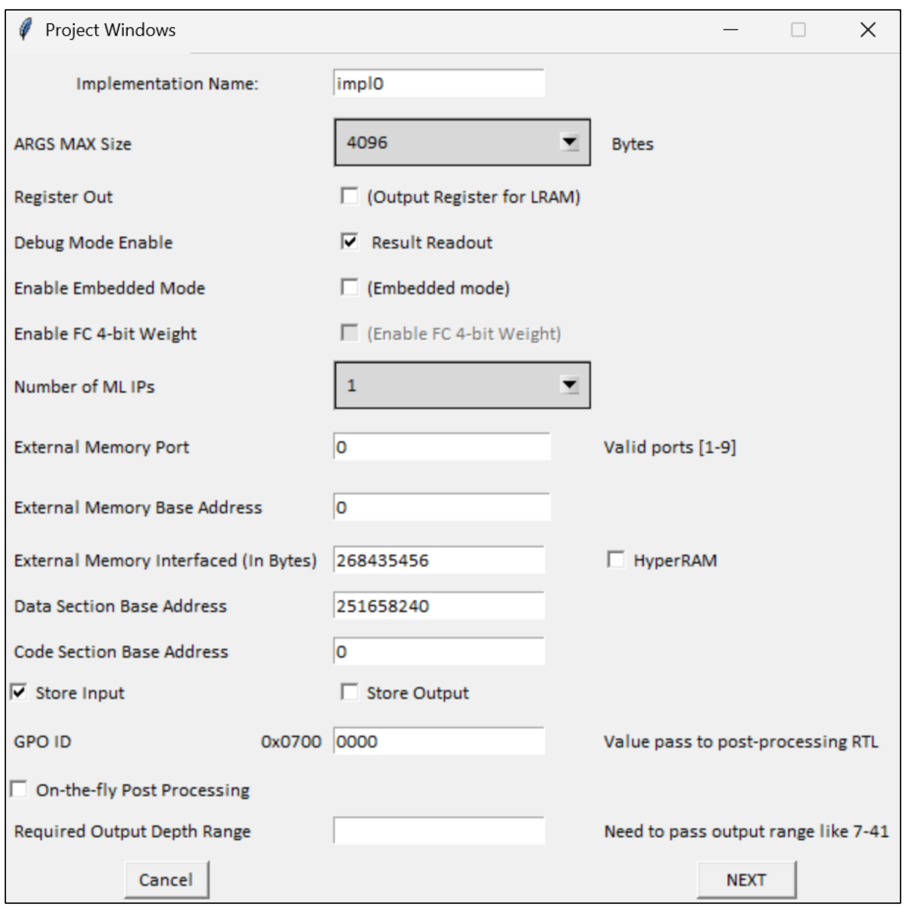


Figure 5.11 Project Implementation Window – Avant Advanced IP Part 1

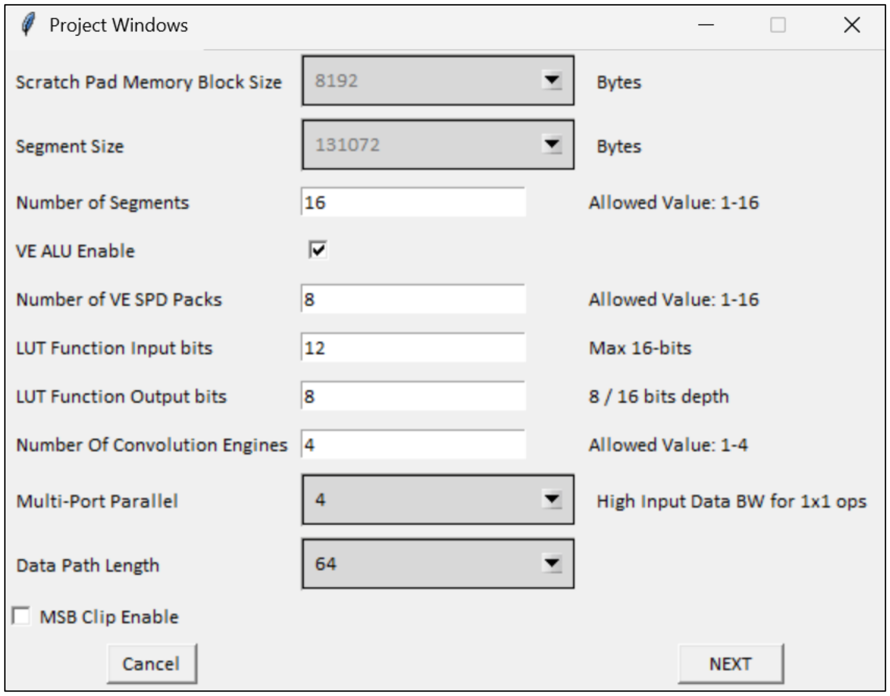


Figure 5.12 Project Implementation Window – Avant Advanced IP Part 2

The settings that are visible and can be adjusted depend on the device, network type, and framework. For example, UltraPlus has a single convolution engine with a fixed size (Figure 5.2), causing those options to be grayed out, while the option for changing your quantization type is only available for BNN projects.

5.1.1. Number of Convolution Engines

You can change the number of convolution engines used by your design, whether they are standard convolution engines or binary convolution engines, to be less than the maximum amount supported on your device. The ability to use less than the maximum depends on the specific device. For example, certain LatticeECP5 products can support up to eight CNN engines, allowing you to reduce your usage. For CertusPro-NX and Avant devices, with Advanced CNN IP 4*N, a number of output channels are generated in parallel. N = 1 for the CertusPro-NX device, and N = 1-4 for Avant devices.

5.1.2. Enable Dual Core Mode

Selecting Enable Dual Core Mode enables dual core mode in ECP5, CrossLink-NX (Optimized, Extended), or CertusPro-NX (Optimized, Extended) devices. When enabled, it uses two DSP blocks per convolution engine. This option is checked and enabled by default. This feature is only supported in ECP5, CrossLink-NX (Optimized, Extended), and CertusPro-NX (Optimized, Extended) devices.

5.1.3. Enable Quad Core Mode

Selecting Enable Quad Core Mode enables quad core mode in CertusPro-NX (optimized) devices. When enabled, it uses four DSP blocks per convolution engine. This option is checked and enabled by default. This feature is only supported in CertusPro-NX (optimized) devices.

5.1.4. On-Chip Memory Block Size

The On-Chip Memory Block size option is only visible for projects targeting iCE40 UltraPlus devices, allowing you to select from three entries from the drop-down menu: 16,384, 32,768, and 65,536. These correspond to three possible memory configurations.

- 16,384 - 16k, 16-bit (32 Kilobyte) Single SPRAM
- 32,768 - 32k, 16-bit (64 Kilobyte) Dual SPRAM
- 65,536 - 64k 16-bit (128 Kilobyte) Quad SPRAM

When using single SPRAM mode, the rest of the memory, over 128 kilobytes, can be used for storing firmware. When using Quad SPRAM, provide external memory for storing firmware.

5.1.5. Number of On-Chip Memory Blocks

The Number of On-Chip Memory Blocks setting specifies the number of discrete blocks in the EBR that are utilized in the DNN Inference Machine. On ECP5 devices, you are required to have a minimum of one plus an additional one for each convolution engine used by your design. For designs using the iCE40 UltraPlus device, the number of blocks is fixed.

5.1.6. Mobilenet Mode for iCE40 UltraPlus, CrossLink-NX Compact, and CertusPro-NX Compact

Mobilenet Mode allows you to select Conv1x1 mode for devices. Three modes, single, dual, and quad, are available to perform 1 × 1 convolutions for iCE40 UltraPlus devices. Quad mode provides the best performance and highest resource. consumption. The single mode is the slowest among the three but uses the least resources.

For CrossLink-NX Compact and CertusPro-NX Compact, only quad mode is available.

5.1.7. Argmax Memory Size

The Argmax Memory Size option allows you to select memory 4k/8k for Argmax pooling metadata, which can be reused while unpooling. This option is available for Extended and Advanced CNN IPs only.

5.1.8. Scratch Memory Size

The Scratch Memory Size option is only visible for projects targeting iCE40 UltraPlus, CrossLink-NX, and CertusPro-NX devices, allowing you to select from two entries in the drop-down menu: 1,024, 2048, 4096, and 8192 based on selected devices. These four options select whether the design uses 1K, 2K, 4K, or 8K of the scratch memory. For iCE40 UltraPlus, the default is 4K and is the recommended setting, though in some cases that require reduced resource utilization, 1K can be selected. Whereas for CrossLink-NX and Certus-NX devices, 8192 is the default and recommended setting. Some designs that utilize less resources may wish to select the other options.

Note: For iCE40 UltraPlus devices, with Quad mode as Conv1x1 mode, all other convolutions (except 1 × 1 convolution) use 2× scratch size. For example, if you select a 2048-byte scratch size internally, 3 × 3 convolutions use 4096-byte scratch memory, and 1 × 1 convolution uses two separate convolutions with a 2048-byte scratch size each.

5.1.9. Debug Mode Enable

This Debug Mode Enable option can be used to enable the write/debug signal on post processing RTL. If unchecked, write mode is enabled; otherwise debug mode is enabled.

5.1.10. Embedded Mode for CrossLink-NX Optimized and CertusPro-NX Optimized

This option is only visible for projects targeting CrossLink-NX Optimized and CertusPro-NX Optimized devices. This option allows you to run your model without using external memory when embedded mode is enabled. Embedded mode also supports branching structures (only residual blocks, not concat structures) and multiple-output networks like single-shot detector (SSD) architectures.

Note: If you observe a memory error, such as a particular layer requiring more memory than the current LRAM size, you can try with a higher LRAM size (for example, QUAD LRAM if currently DUAL LRAM is being used). If it is not possible, reduce the filter or dimension. To run the same model, turn off embedded mode so the tool can use external memory.

5.1.11. Input Memory Assignment

This setting specifies which EBR memory blocks should be used to store input data in cases where specific memory blocks should be used. The values must be comma-separated. For example, "1, 2" specifies that EBR 1 and 2 should be used. If left blank, the software automatically assigns memory blocks.

5.1.12. Output Memory Assignment

Similar to input memory assignment, the output memory assignment setting identifies which EBR should be used when specified and is automatically assigned when left blank.

5.1.13. Off-Chip Data Memory Start Address

This setting determines the memory address in DRAM where the convolution design starts storing and loading data. The amount of DRAM required depends on your neural network and your EBR settings, with larger networks or implementations with lower EBR usage requiring more DRAM. If you intend to read or write input or output to a memory location, you must have storage enabled, while having it disabled requires you to provide input and output from something external to the provided IP block.

Do Not Use (ECP5 Only)

The Do Not Use option disables all DRAM usage. In addition to not storing the input or output in DRAM, it also disables the ability to store data from intermediate stages in the DRAM. This mode may not be compatible with all networks.

Store Input

Enabling Store Input indicates that external memory (HyperRAM/DRAM) is used for input rather than another source. Disabling this setting prevents external memory from being used to store input. In this case, you need another way of providing input into your design.

Store Output

Similar to Store Input, the Store Output option indicates that external memory (HyperRAM/DRAM) is used for output rather than another source.

5.1.14. Collapse Layer

The Collapse Layer option enables you to merge the layers Convolution, BatchNorm, and Scale during the Compile and Simulation stages, implementing them as a single Convolution layer in hardware. This feature is applicable for networks with convolution, batch norm, and scale layer architectures. Designs using this optimization should see a reduction in scale cycles, and a possible reduction in memory access cycles.

5.1.15. Data Preprocessing

The supported preprocessing operations are shifting (mean), scaling (scale), and resizing. For demo designs, some preprocessing has already been applied to the hardware. Refer to the IP documentation for details on preprocessing in a specific design.

Input data scaling can be implemented in the firmware. The stored_frac bit is adjusted to scale the input data. For more information, see the Lattice sensAI Human Counting AI Demo web page where the input image is scaled from 0-255 to 0-2 in the firmware by setting stored_frac bits to 1.7 in sensAI.

Note: Shifting (mean) preprocessing must be done in the preprocessing RTL, not in the sensAI firmware. It is included in the user interface for testing purposes, but the final implementation must perform mean preprocessing in the RTL design and set the mean to 0 in sensAI. The iCE40 UltraPlus device does not support scaling in sensAI; however, scaling and resizing are supported in sensAI.

For example, consider an input image with a range of 0–255, a scale of 0.0078125, and a mean of 128. The input data range is from –1 to 1. When the firmware is generated, only scaling is performed using the stored_frac value in sensAI, resulting in a range of 0–1. This occurs because the signed format (0.7) in stored_frac is not being shifted. Perform shifting in the preprocessing RTL to implement the mean. To bypass mean/scale preprocessing, use the default values: mean = 0 and scale = 1.0.

For designs with input image data, preprocessing can be managed in sensAI source files. In TensorFlow and Keras, preprocessing can be added using additional node operations.

For a given mean and scale, the final output fed to the network is:

Output Pixel = (Input Pixel – Mean) × Scale

Mean subtraction is always performed before scaling. The mean value is an integer, and the scale value is a float.

For a better understanding of how sensAI (not the firmware) calculates ranges, see the following examples:

- Input image pixel range is 0–255; mean: 128; scale: 1/256 (0.00390625):
  - Output pixel range is: –0.5 to 0.5.
- Input image pixel range is 0–255; mean: 0 (default value); scale: 1/256 (0.00390625):
  - Output pixel range is: 0 to 1.
- Input image pixel range is 0–255; mean: 128; scale: 1.0 (default value)
  - Output pixel range is: –128 to 127.
- Input image pixel range is 0–255; mean: 0 (default value); scale: 1/128 (0.0078125)
  - Output pixel range is: 0 to 2

The final type of preprocessing is resizing. Resizing is required, and the input image is automatically resized into the input data blob using an interpolation function. This step cannot be bypassed.

Mean Value for Data Pre-Processing

The mean value is used to normalize input data. You must specify a value or use the default setting. If you use a value other than the default, specify it in this setting. It is not derived from your neural network files. The mean value is subtractive. For example, a mean value of 1 subtracts 1 from all results. The default is 0, which does not affect the output. As noted in the previous section, the final network implementation must perform mean preprocessing in the RTL design. Set the mean to 0 in sensAI.

Scale Value for Data Pre-Processing

The scale value is used to scale data values. You must specify a value or use the default setting. If you use a value other than the default, specify it in this setting. It is not derived from your neural network files. The scale value is multiplicative. For example, a mean value of 0.5 multiplies all results by 0.5. The default is 1, which does not affect the output. The maximum scale value supported by sensAI without additional RTL preprocessing is 1.0. For this reason, scaling is recommended in the preprocessing RTL in most cases.

When using both scale and mean values, note that the mean is subtracted first, then the scale is applied.

Output Pixel = (Input Pixel – Mean) x Scale

Note: If scaling is handled in your preprocessing RTL, set it to 1.0 in sensAI.

5.1.16. GPO ID

The GPIO ID option is available for communication from firmware to outside blocks. The total value of the GPO ID is 32 bits. The first 16 bits are fixed and indicate the sensAI tool version. You can configure the last 16 bits.

5.1.17. On the Fly Post Processing

The On-the-Fly-Post-Processing option is available for iCE40 UltraPlus, CrossLink-NX, CertusPro-NX, and Avant devices only. Readout single data at a time for on-the-fly post-processing of the result without storing complete output on the post-processing side RTL. It is only applicable to detection-type of networks. It is useful for reducing on-chip memory utilization in post-processing RTL. The expected output depths are shown below in order for the N class.

Conf [1depth/anchor]	class prob[N depth/anchor]	Bbox [4 depth x,y,w,h / anchor]

Figure 5.13. On-the-Fly Post Processing Format

Select the on-the-fly post processing checkbox and provide the number of classes in the number of classes for detection field. The number of anchors and grid dimension are calculated using the dimension of the output and the number of classes provided by the user, as follows:

If output dimension is (D,H,W) and number of classes are N: then

Number of anchors = D//(conf + class probabilities + (x,y,h,w)) = D // (1 + N + 4)

And grid size = H x W

For example, if the number of classes for detection is 2, then the NNC compiler will postprocess thed data flow with a single anchor and grid as per the below order and repeat it for all other results.

Confidence	Class – 0	Class – 1	X – Offset	Y – Offset	W – Offset	H - Offset

Figure 5.14. On-the-Fly Post Processing Data Flow

5.1.18. Required Output Depth Range

The option is available for iCE40 UltraPlus, CrossLink-NX, CertusPro-NX, and Avant devices only. If the last layer in a network is a convolution layer, this option allows for only processing selected filters from that convolution layer. This sets weight_slice, i_weight_slice, and output_data_length values in the ***.yml*** file at the time of analysis.

For example, if the required 'output depth range' value is '7-13', then it processes only the 7^{th} to 13^{th} filters (including the 13^{th}) and stores the output at the output address.

5.1.19. Sample Rate for Data Pre-Processing

If the input data is audio data (.wav), this option is displayed in the implementation window. This feature reflects the sample rate of audio data. The equation used for audio preprocessing is: window_duration = (network_input_dimension/sample_rate) * down_sampling. The following example demonstrates this.

Sample Rate

```
Network_input_dimension = [1,1,8320,1], sample_rate = 8000 ,down_sampling = 1
                window_duration = (8320/8000)*1
                window_duration =  1.04
```

5.1.20. Down Sampling for Data Pre-Processing

If the input data is audio data (.wav format), this option is displayed in the implementation window. This feature samples the audio data.

5.1.21. On-Chip Large Memory Size

CrossLink-NX, CertusPro-NX, and Avant devices only. This option selects the size of the Large Random-Access Memory (LRAM) block available. For Crosslink-NX and CertusPro-NX devices and IP other than Advanced IP, this option allows you to select from three entries from the drop-down menu: 65,536, 131,072, and 262,144 (Quad LRAM). These correspond to two possible IP-dependent memory configurations:

- 65,536 - 0.5 megabytes (16384 x 32)
- 131,072 - 1 megabyte (32768 x 32)
- 262144 - 2 megabytes (65536 x 32)

For Advanced IP, you can select the size of Large Random-Access Memory(LRAM) by giving the number of segments. For Advanced IP with a 32-bit data path, each segment size is 65,536 bytes, and with a 64-bit data path, the segment size is 131072 bytes.

For Certus-Pro devices with Advanced IP, the range of segments you can choose from is from 1 to 7. For Avant Device, you can choose segment numbers from 1 to 16.

5.1.22. External Memory Interfaced (In Bytes)

CrossLink-NX, CertusPro-NX, and Avant devices only. This option specifies the size of the external memory in bytes.

HyperRAM

This option enables addressing HyperRAM rather than DRAM for external memory. HyperRAM is enabled by default, but designs for setups that do not utilize HyperRAM wish to disable this feature.

5.1.23. Code Section Base Address

CrossLink-NX, CertusPro-NX, and Avant devices only. This setting determines the memory address in external memory where the firmware is stored.

5.1.24. Register Out

CrossLink-NX devices only. This parameter in the GUI is equivalent to the LRAM_OREG configuration parameter in Optimized CNN and Compact CNN IP [Crosslink-NX device].

For Crosslink-NX device:

- Register Out is Unchecked: Do not use the output register for LRAM. The firmware will be backward compatible, and it can be utilized with older IPs.
- Register Out is Checked: If you use the output register option for LRAM, NNC will generate ML firmware to compensate for the latency produced by registering the output of LRAM.

Using the output register option in CNN IP for LRAM will provide better timing with less than 1% cycle degradation.

For the CertusPro-NX device, the output register is always used for LRAM, and by default, NNC generates proper firmware to compensate for the latency of that device.

5.1.25. Data Section Base Address

CrossLink-NX, ECP5, CertusPro-NX, and Avant devices only. This setting determines the memory address in external memory where the convolution design is to be stored and loaded.

For example, below are the default memory sizes in ECP5 DRAM:

- code section size is 240MB – (0 to 251658240/0xF000000)
- data section size is 16MB – (251658240/0xF000000 to 268435456/0x10000000)
- data section base address – 251658240/0xF000000

By changing the data section base address to lower values, you can increase the memory allocated for data (the same amount of memory allocated for code is decreased). To allocate 48MB to the data section, the data section base address should be 218103808 (0xD000000).

- code section size 208MB (256-48) – (0 to 218103808/0xD000000)
- data section size 48MB - (218103808/0xD000000 to 268435456/0x10000000)
- data section base address - 218103808/0xD000000

5.1.26. Number of Segments

For CertusPro-NX and Avant devices, Advanced IP only. This setting determines the total lram size available. Valid values range from 1 to 7 for CPNX Advanced and 1 to 16 for Avant Advanced. LRAM size will be equal to (number of segments x segment size). The default value of the number of segments is 16 for advanced.

5.1.27. Segment Size

For CertusPro-NX and Avant devices, advanced CNN IP is only available. This setting determines the segment size, which, along with the number of segments, determines the LRAM size. For the CPNX device, the advanced IP segment size is fixed to 65536 bytes. For the Avant device, if 64-bit data path mode is selected, segment sides will be 131072 each.

5.1.28. VE ALU Enable

For CertusPro-NX and Avant devices, this option is available only for advanced CNN IP. This setting allows the user to enable or disable the VE ALU engine, which is use for elementwise Add/Mul/Pow and similar operations, but at the cost of extra resources.

If the model includes only the Add operation, there is no need to increase VE ALU engine resources because Add is supported by the depthwise 1x1 engine. Therefore, the VE ALU engine can be disabled to save on resources.

5.1.29. Number of VE SPD

For CertusPro-NX and Avant devices, advanced CNN IP is only available. This setting determines the number of VE spd, for 1x1 and Eltwise addition operations. The valid value ranges from 1 to 8. The default value is 8.

5.1.30. Multiport Parallel

For CertusPro-NX, advanced IP, and Avant devices only. This setting determines the input data bandwidth for 1x1 operations. A parallel port will speed up the execution of 1x1 operations, but at the cost of increased resource utilization.

5.1.31. Datapath Width

This setting is only available for Avant devices and advanced IP. This setting determines the width of the data path inside the IP. As the data path width increases, more bytes will be transferred in each memory transaction cycle.

5.1.32. LUT Input Bits

Setting for the input bits of the LUT used in the sigmoid, SiLu, softmax or DivNoNan function. The input ranges from 5 to 12 bits.

5.1.33. LUT Output Bits

Specifies the output bits of the LUT used in the sigmoid, SiLu, softmax or DivNoNan function. The output can be either 8 or 16 bits, depending on whether it is an intermediate or final layer.

5.1.34. LUT MSB Clip

Clip MSB from the number of LUT input bits. If function output saturates on both higher and lower values of input, we can consider those saturating values as constant and clip the MSB if input bits for less resource utilization by LUT and also better performance, and now LUT instead of k bits of input uses k-1 bits.

5.1.35. Create Quantized Version

If the input model is not quantized, enabling this option generates a quantized version of the input model. The compiler tool includes the QuantReLU node after every ReLU node and generates the model, which will be used for further network compilation processing. When generating the quantized version of the input model, validation data can be provided by specifying the Validation Datapath so that the compiler uses validation data when creating the quantized model. If the input model is already quantized, enabling this option has no impact. For a partially quantized input model, the tool gives an error.

5.1.36. Validation Datapath

Specify the path to the validation data. Validation data is used when creating the quantized version of the input model.

5.1.37. Enable FC 4 Bit Weight

Enable FC weights in 4 bits data width while performing FC computations in engine. While training the model, use learned step quantization and 4 bits for the Dense/Fully Connected layer. When providing the trained model as input to the compiler, enable this flag to indicate to the compiler that this feature is active.

This feature is available only for the Optimized IP.

5.1.38. Number of ML IPs

Number of ML IPs used to run the network. Default is 1.

5.1.39. External Memory Port

From this given external memory port number, the logical address for the interfaced external LPDDR4 memory is derived.

5.1.40. Initial LPDDR4 Address

This is used for USB debugging and to convert logical address into physical address.

5.2. Quantization

5.2.1. Learned Step Quantization (LSQ)

This quantization method is based on the paper Learned Step Size Quantization. In this approach, the floating-point step size is learned during training to represent weights and activation data in low precision. The method performs computations such as convolution and fully connected layers in low precision, then retrieves the high-precision output using the learned step sizes. In SensAI, LSQ is used for convolution, fully connected, matrix multiplication (Matmul), and elementwise (Eltwise) addition operations in integer format.

Training Learned Step Quantization Model Using Lscquant Package

Models can be trained with Learned Step Quantization using the Lscquant package available on the Lattice website. You can train models with 8-bit or 4-bit learned step quantization using the schemes provided in the package. For more details, refer to the document included with the Lscquant package. For a reference model trained with LSQ, see the example in the MV2 (MobileNet V2) section. The current compiler version supports only the following schemes from the Lscquant package:

- LSQ_CONV8_ACT8U_DENSE8_OUT16S
- LSQ_CONV8_ACT8U_DENSE4S_OUT16S
- LSQ_CONV8_ACT4U_DENSE8_OUT16S
- LSQ_CONV8_ACT4U_DENSE4S_OUT16S

To train a model with LSQ, use the custom layers discussed here and set quantization="lsq" when creating the neural network. These custom layers are also compatible with arguments from Keras base classes. The following layers are defined in the Lscquant package:

Lscquant.layers.QuantizeConv2D(do_quant_bias=False, quantization="lsq", bits=8, range_min=None, range_max=None, **kwargs)

- Supports 3 x 3 and 1 x 1 convolution layers.
- Always set per_channel_quant = False.
- Derived from tensorflow.keras.layers.Conv2D; kwargs includes all arguments supported by the base class.

Lscquant.layers.QuantizeDepthwise2D(do_quant_bias=False, quantization="lsq", bits=8, range_min=None, range_max=None, **kwargs)

- Supports 3 x 3 depthwise convolution layers.
- Always set per_channel_quant = False.
- Derived from tensorflow.keras.layers.DepthwiseConv2D; kwargs includes all arguments supported by the base class.

Lscquant.layers.QuantizeBatchNormalization(quantization="lsq", bits=8, range_min=None, range_max=None, **kwargs)

- Batch normalization layer.
- Derived from tensorflow.keras.layers.BatchNormalization; kwargs includes all arguments supported by the base class.

Lscquant.layers.QuantizeDense(do_quant_bias=False, quantization="lsq", bits=8, range_min=None, range_max=None, **kwargs)

- Dense (inner product or fully connected) layer.
- Always set per_channel_quant = False.
- Derived from tensorflow.keras.layers.Dense; kwargs includes all arguments supported by the base class.

Lscquant.layers.QuantizeAdd(quantization="lsq", bits=8, is_signed=False, range_min=None, range_max=None, **kwargs)

- Elementwise (Eltwise) addition layer.
- Derived from tensorflow.keras.layers.Add; kwargs includes all arguments supported by the base class.

Lscquant.layers.QuantizeActivation(activation="relu", quantization="lsq", bits=8, range_min=None, range_max=None, **kwargs)

- Derived from tensorflow.keras.layers.Activation; kwargs includes all arguments supported by the base class.

Lscquant.layers.QuantizeConcat(axis=-1, quantization="lsq", **kwargs)

- Derived from tensorflow.keras.layers.Concatenate; kwargs include all arguments supported by the base class.

Lscquant.layers.QuantizeOutput(quantization="lsq", bits=16, is_signed=False, range_min=None, range_max=None, step_size=1/1024, **kwargs)

- Derived from tensorflow.keras.layers.Layer; kwargs includes all arguments supported by the base class.
- Fixed step size of 1/1,024 produces a quantized output in the Q5.10 fixed-point representation format.

Lscquant.layers.FocusLayer(focus_kernel_size=(2, 2) , **kwargs)

- Derived from tensorflow.keras.layers.Layer; kwargs includes all arguments supported by the base class.
- focus_kernel_size is a two-dimensional tuple that specifes the vertical and horizontal strides.

Lscquant.layers.Split(focus_kernel_size=(2, 2) , **kwargs)

- Derived from tensorflow.keras.layers.Layer; kwargs includes all arguments supported by the base class.
- Performs a channel-wise split of the input tensor using tf.strided_slice with begin_mask and end_mask attributes.

Quantizing Keras Model Using Schemes

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, Lambda
from tensorflow.keras import Model, Input
from tensorflow.keras import backend as K
import lscquant

def create_model():
    ip = Input(shape=(64,64,3))
    x = Conv2D(filters=4, kernel_size=3, strides=1, padding="same")(ip)
    x = BatchNormalization()(x)
    out = ReLU()(x)
    model = Model(inputs=ip, outputs=out)
    return model


# creating model without quantization
model = create_model()


# selecting schemes from lscquant package
scheme = 'lsq-default'

# generating quantized version of model
lsq_model = lscquant.model.build.build_quantization_model(model, scheme)
```

The Lscquant package provides various schemes for model quantization. Shown here is the lsq-default scheme, which quantizes activation and weights to 8 bits.

After creating the model using native Keras functions such as Conv2D, Depthwise2D, Add, ReLU, Dense, and Concat, call the build_quantization_model( ) function from the Lscquant package to apply an available quantization scheme. For more details, refer to the Lscquant package documentation.

Post Training Quantization with Learned Step Quantization

Post-training quantization provides a conversion method that reduces model size while improving hardware response times. This process quantizes a pre-trained floating-point TensorFlow or Keras model.

When you provide the float model as input in SensAI, enable the *Create Quantized Version* option in the Project Window (see Figure 5.15) to apply post-training quantization. This generates a quantized version of the input model. The step sizes of the PTQ model (post-training quantized model) are dynamic, similar to those in learned step quantization. However, these step sizes are calculated from validation data provide through the path specified in the validation data path option. If validation data is not provided, parameters are derived from the single input image selected during project creation.

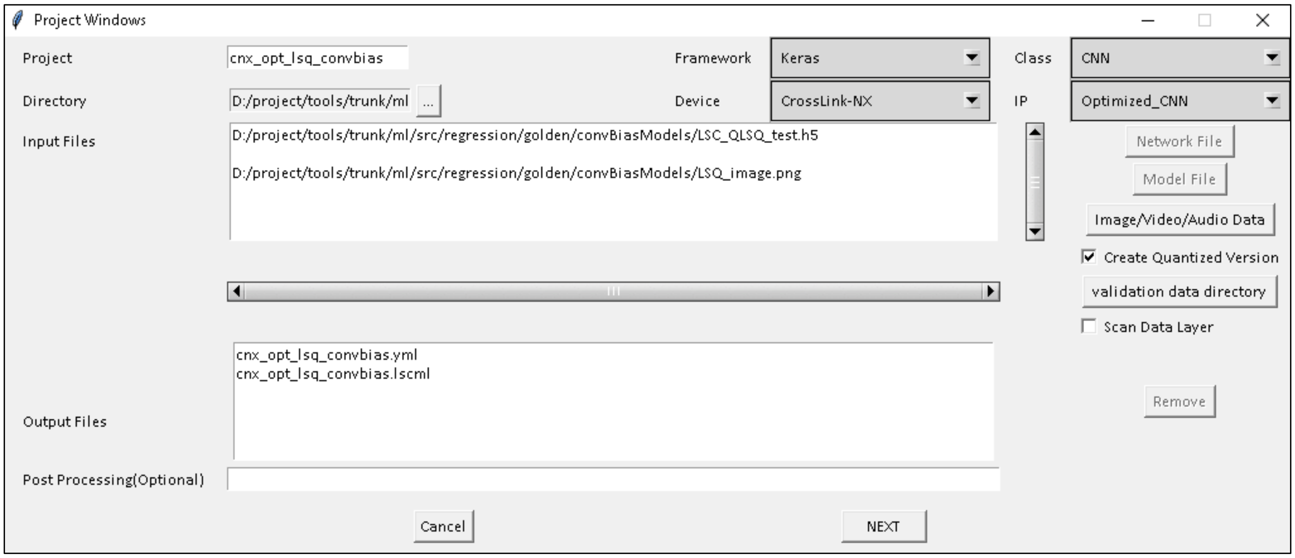


Figure 5.15. Create Quantized Version Flag

The following table summarizes the Learned Step Quantization support provided by the SensAI stack across different Lattice devices.

Table 5.1. Learned Step Quantization Details with Device Type

Quantization	Type	Device		
		ECP5	iCE40 UltraPlus	CrossLink-NX, CertusPro-NX, Avant
Activation	16b	Not supported	Not supported	Not supported
	8b			Requires Quantization aware training
	4b			Only supported in Optimized IP and model with Learned Step Quantization.
Weights	16b	Not supported	Not supported	Not supported
	8b			Requires Quantization aware training
	4b			Only supported for Fully Connected layer in CrossLink-NX device and Optimized IP and model with Learned Step Quantization.

5.2.2. Fixed Point Quantization (FPQ)

The data in sensAI can be quantized using the QuantReLU layer in Caffe or the predefined quantization function in TensorFlow to perform quantization on unsigned 8-bit activation data in the training phase. Neural Network Compiler 7.0 only supports using 8-bit data to represent quantized data.

SensAI automatically calculates the number of fraction bits and decimal bits needed to store the quantized data, which can be found in the stored_frac section of the report panel in the main window. If you would like to quantize the activation data yourself, for example, with min = 0.0 and max = 2.0, then use the 8-bit calculation to take place (after the ReLU layer) as follows:

Neural Network Compiler dedicates 0 bits for signs (all positive values), 1 bit for decimal, and 7 bits for fractions, resulting in the representation of data in hardware having a range of 0.0 to 1.9921875. You should use a maximum range that is a power of 2 (5 or 7 values), as there is no dedicated hardware for quantization. The following tables show the ranges that are powers of two and the respective fraction bits and decimal bits.

Fixed Point Quantization Using Lscquant Package

Models can be trained using Fixed 8b Quantization with the Lscquant package provided at the Lattice website. You can train models using the different schemes available in the package. For more information, refer to the document provided with the Lscquant package and the example provided in the MV1 (MobileNet V1) section.

The following is a code snippet for training the fixed-point quantized model using the Lscquant package.

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, Lambda
from tensorflow.keras import Model, Input
from tensorflow.keras import backend as K
import lscquant

def create_model():
    ip = Input(shape=(64,64,3))
    x = Conv2D(filters=4, kernel_size=3, strides=1, padding="same")(ip)
    x = BatchNormalization()(x)
    out = ReLU()(x)
    model = Model(inputs=ip, outputs=out)
    return model


# creating model without quantization
model = create_model()


# selecting schemes from lscquant package
scheme = 'fpq-default'

# generating fixed point quantized version of model
fpq_model = lscquant.model.build.build_quantization_model(model, scheme)
```

The following tables show that increasing the quantization range results in the data representation becoming less accurate. For this reason, the suggested range is 0 to 2.

Table 5.2. Unsigned 8-Bit Quantization (Fixed Point Quantization)

Unsigned 8-Bit						
Min (Protofile)	**Max (Protofile)**	**Sign Bits**	**Decimal Bits**	**Fraction Bits**	**Min (Hardware)**	**Max (Hardware)**
0	1	0	0	8	0	0.99609375
0	2	0	1	7	0	1.992188
0	4	0	2	6	0	3.984375
0	8	0	3	5	0	7.96875
0	16	0	4	4	0	15.9375

Table 5.3. Signed 8-Bit Quantization (Fixed Point Quantization)

Signed 8-Bit						
Min (Protofile)	**Max (Protofile)**	**Sign Bits**	**Decimal Bits**	**Fraction Bits**	**Min (Hardware)**	**Max (Hardware)**
–2	2	1	1	6	–1.98438	1.984375
–4	4	1	2	5	–3.96875	3.96875
–8	8	1	3	4	–7.9375	7.9375

The following table summarizes the Fixed-Point Quantization support provided by the SensAI stack across different Lattice devices.

Table 5.4. Fixed Point Quantization Details with Device Type

Quantization	Type*	Device		
		ECP5	**iCE40 UltraPlus**	**CrossLink-NX, CertusPro-NX, and Avant**
Activation	16b	Default- Post processing Quantization in tool	Default- Post processing Quantization in tool	Not supported
	8b	Quantization-aware training is required	Quantization-aware training is required	Quantization-aware training is required
	4b	Not supported	Not supported	Only supported in Optimized IP and model with Learned Step Quantization.
Weights	16b	Default- Post processing Quantization in tool	Default- Post processing Quantization in tool	Not supported
	8b	Not supported	Quantization-aware training is required	Quantization-aware training is required

*__Note__: Except for the above-mentioned type, the Lattice sensAI stack does support 1b [BNN] and 4b quantization. Contact Lattice representatives to get more information.

As seen in Table 5.4, the NNC compiler internally uses the default 16b for representing data if no supported 8b quantization structure is used in the input network [except image input; the NN compiler always uses 8b for the input image].

Note: The quantization techniques is one of the best optimization techniques available in the market, and we always recommend users use the provided quantization techniques and functions for better performance in terms of FPS and power consumption.

Table 5.5 provides layer-wise support for quantization.

Table 5.5 Quantization Support in Layers

Layer Type	Quantization Support
Convolution layer	The user can train with the following: • 8b Fixed Point Quantization • 8b Learned Step Quantization • 4b Learned Step Quantization We generally support a –0.5 to +0.5 data range for convolution layer weight quantization, and input to convolution can be 16b or 8b quantized. For 8b activation quantization, the generally supported range is 0 to 2.
MaxPooling or AveragePooling or ResizeBilinear	Input data type should be equal to output datatype.
Batch norm layer	Do not use any type of quantization for a better learning of model.
Fully Connected layer	The user can train with 8b quantization. We generally support a –0.5 to +0.5 data range for Fully Connected layer weight quantization, and input to Fully Connected layer can be 16b or 8b quantized. 4-bit input is not supported for Fully Connected layer.
Eltwise Layer	Input data type should be equal to output data type, i.e., if output has been quantized to 8b, then both inputs should be in 8b quantized format.
ReLU or LeakyReLU	There is no dependency on the input type.

Note: If the model is trained with LSQ, provide the trained keras model .h5 as input to the SensAI Neural Network Compiler instead of converting it to TensorFlow .pb or .onnx format.

5.2.3. Fixed Point Quantization Training in Caffe

In Caffe, fixed point quantization can be implemented with the QuantReLU layer. The following example demonstrates how the layer is used.

Caffe QuantReLU Layer

```
layer {
  name: "fire1/div"
  type: "QuantReLU"
  bottom: "Scale1"
  top: "Scale1"
  quantize_param {
    num_bit: 8
    min: 0.0
    max: 2.0
    resolution: 256.0
  }
}
```

5.2.4. Fixed Point Quantization Training in TensorFlow

For TensorFlow, fixed point quantization can be implemented using the quantization function.

TensorFlow Quantization Function

```
def lin_8b_quant(w, min_rng=-0.5, max_rng=0.5):
    min_clip = tf.rint(min_rng*256/(max_rng-min_rng))
    max_clip = tf.rint(max_rng*256/(max_rng-min_rng))

    wq = 256.0 * w / (max_rng - min_rng)              # to expand [min, max] to [-128,
128]
    wq = tf.rint(wq)                                          # integer (quantization)
    wq = tf.clip_by_value(wq, min_clip, max_clip)     # fit into 256 linear
quantization
    wq = wq / 256.0 * (max_rng - min_rng)             # back to quantized real number,
not integer
    wclip = tf.clip_by_value(w, min_rng, max_rng)     # linear value w/ clipping
    return wclip + tf.stop_gradient(wq – wclip)
```

The corresponding Tensor graph resembles the Figure 5.16.

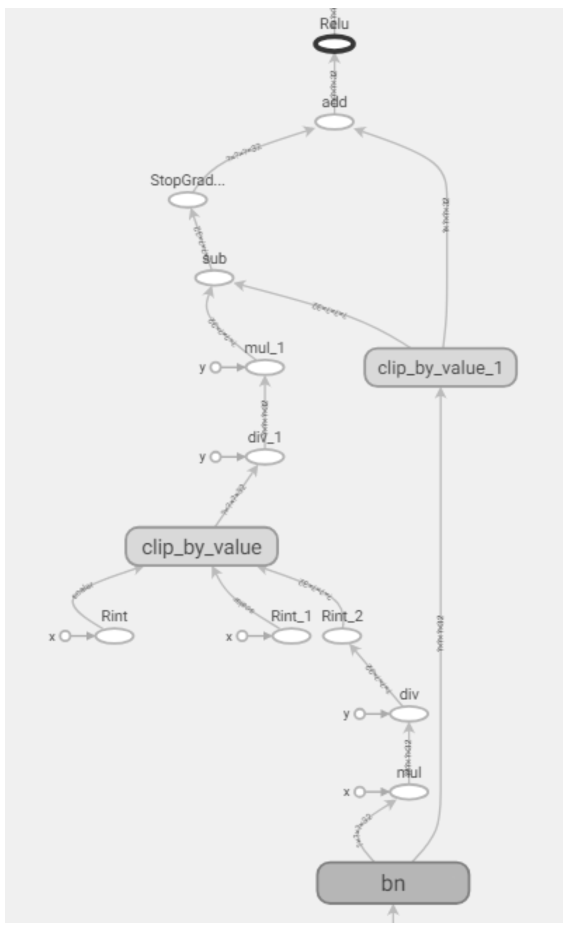


Figure 5.16. Tensor Graph Quantization Nodes

5.2.5. Fixed Point Quantization Training in Keras

8-bit activation quantization can be done by using a Lambda layer from tf.keras.layers, and weight quantization can be done using kernel constraints. Both methods are explained in the snippet below.

Keras Fixed Point Quantization Function

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, Lambda
from tensorflow.keras import Model, Input
from tensorflow.keras import backend as K

def lin_8b_quant(w, min_rng=-0.5, max_rng=0.5): ## 8-bit activation quantization in Keras using Lambda
layer
    if min_rng==0.0 and max_rng==2.0:
        min_clip = 0
        max_clip = 255
    else:
        min_clip = -128
        max_clip = 127
    wq = 256.0 * w / (max_rng - min_rng)              # to expand [min, max] to [-128, 128]
    wq = K.round(wq)                                  # integer (quantization)
    wq = K.clip(wq, min_clip, max_clip)     # fit into 256 linear quantization
    wq = wq / 256.0 * (max_rng - min_rng)             # back to quantized real number, not integer
    wclip = K.clip(w, min_rng, max_rng)     # linear value w/ clipping
    return wclip + K.stop_gradient(wq - wclip)

class MyConstraints(tf.keras.constraints.Constraint): ##Used for 8-bit weight quantization is Keras
    def __init__(self,name="",  **kwargs):
        super(MyConstraints, self).__init__(**kwargs)
        self.name=name
    def __call__(self, w):
        with tf.compat.v1.variable_scope(self.name + "_CONSTRIANTS") as scope:
            return lin_8b_quant(w)


    def get_config(self):
        return {"name":self.name}

def act_quant_8b(x, a_bin=16, min_rng=0.0, max_rng=2.0): # For use in Lambda layer

    x_quant = lin_8b_quant(x, min_rng=min_rng, max_rng=max_rng)
    return x_quant

def create_model():
    ip = Input(shape=(64,64,3))
    x = Conv2D(filters=4, kernel_size=3, strides=1, padding="same", activation='linear',\
                    kernel_constraint=MyConstraints("conv2d_1"),use_bias=False)(ip) ## Using Kernel
constraints here gets us 8b
                                                                               ## weight quantization
    x = BatchNormalization()(x)
    out = Lambda(act_quant_8b)(x) ##Activation Quantization
    model = Model(inputs=ip, outputs=out)
    return model

create_model()
```

5.2.6. Fixed Point Quantization Training in AutoKeras

8-bit activation and weight quantization are supported in AutoKeras customized layers (similar to the ones in Keras). The user can enable the flags ***quantrelu*** (for activation) and ***kernel_quant*** (for weight) for quantization. AutoKeras custom layers support both quantized and non-quantized models to support all the devices supported by NNC. Please refer to the AutoKeras Reference Design script to use the AutoKeras quantization.

5.2.7. Fixed Point Quantization for iCE40 UltraPlus, CrossLink-NX, CertusPro-NX, and Avant

The Neural Network Compiler 7.0 UltraPlus IP, 4.0 CrossLink-NX IP, and 4.0 CertusPro-NX IP are created by considering input/output data quantization with a range of [0, 2] (2 is non-inclusive, and it is represented in 1.7 fractional format) and a weight quantization range of [-0.5, +0.5](+0.5 is non-inclusive). You must train your network using the quantization function. After training your network in this way, you cannot manually adjust your fractions afterwards in sensAI. The output of all CNN models for UltraPlus in Neural Network Compiler 7.0 is in signed 16-bit format, represented in 5.10 fractional format.

Note that while training models, you must use quantization for all the activations simultaneously. A single data activation is interpreted as all the activations being quantized. This also applies for weight quantization.

Weight quantization is supported in the Keras and TensorFlow platforms, and a script is provided for your use. This script, shown below for convenience, can be used to perform the data and weight quantization.

TensorFlow Data and Weight Quantization for iCE40 UltraPlus

```
#This code is taken directly from the TensorFlow script, w is a tensor here
def lin_8b_quant(  w, min_rng=-0.5, max_rng=0.5,res=256 , offset=-1):

    with tf.Session() as sess:
        min_clip = tf.rint(min_rng*res/(max_rng-min_rng))
        max_clip = tf.rint(max_rng*res/(max_rng-min_rng)) + offset # 127, 255

        wq = (1.0*res) * w / (max_rng - min_rng)              # to expand [min, max] to [-
128, 128]
        wq = tf.rint(wq)                                    # integer (quantization)
        wq = tf.clip_by_value(wq, min_clip, max_clip)     # fit into 256 linear
quantization
        wq = wq /(1.0* res) * (max_rng - min_rng)             # back to quantized real
number, not integer
        wclip = tf.clip_by_value(w, min_rng, max_rng)     # linear value w/ clipping
        qw=sess.run(wclip + tf.stop_gradient(wq - wclip) )
        sess.close()
        #print( qw )
    return    qw
```

The quantization of the activation data is represented in Figure 5.17.

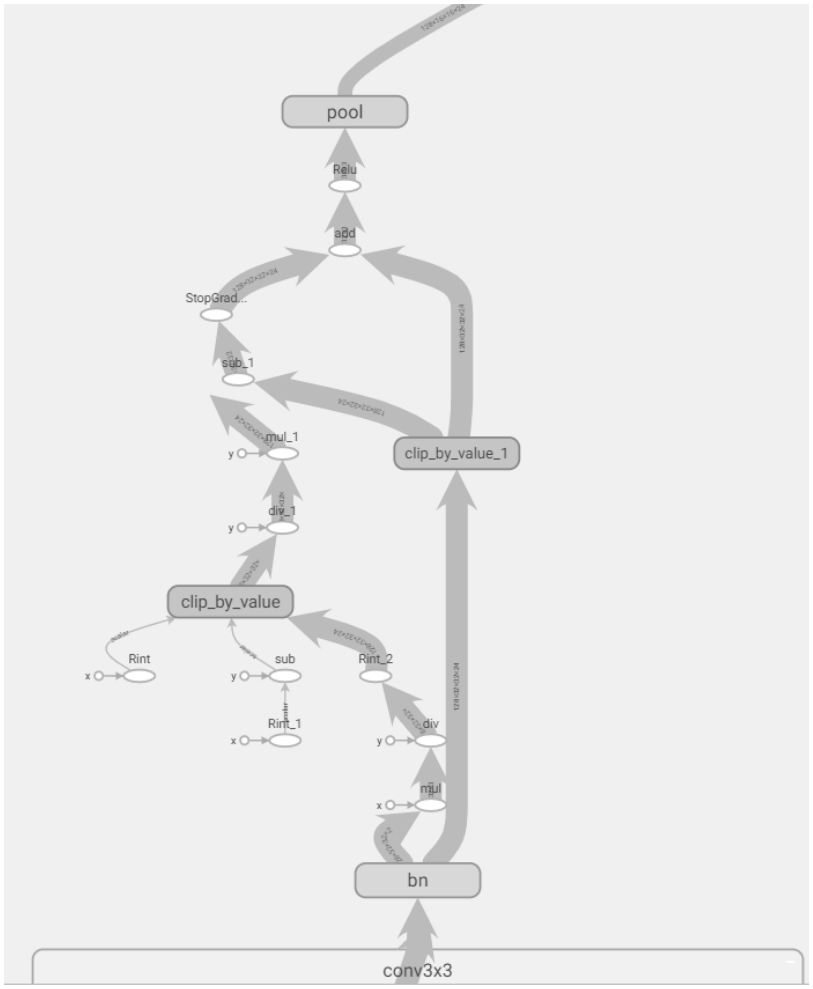


Figure 5.17. Activation Data Quantization Nodes

5.2.8. Fixed Point Quantization Requirements and Suggestions

The following are further requirements and suggestions for fixed point quantization. Consult this list to troubleshoot your designs.

- Always use the collapse layer option when using quantization for ECP5.
- When using Caffe, always use an in-place QuantReLU layer before ReLU activation and after a Batchnorm layer.
- The input Blob is always considered an 8-bit signed/unsigned type if the decimal range of the input data is less than or equal to 256. You can force the use of the 16-bit signed type by overriding the value in stored_frac for the input blob in your report window. Supported formats are 15.0 for 16-bit signed, 8.0 for 8-bit unsigned, and 7.0 for 8-bit signed.

Learned Step Quantization (LSQ) is supported only in Advanced IP.

5.3. Optimization Modes

5.3.1. Mobilenet Mode for ECP5

When creating or modifying a project, ECP5-targeted designs can enable Mobilenet mode to target designs intended to run on the Convolutional Neural Network (CNN) Mobilenet Accelerator IP that has been generated in Mobilenet mode. Unlike the default configuration, the Mobilenet mode is optimized to run Mobilenet designs by implementing the Depthwise and 1×1 Convolution engines in place of some of the standard Convolution engines. This mode is configured to use eight convolution engines, eight Depthwise Convolution engines, and 64 1×1 Convolution engines. Additionally, it always uses 16 EBRs in this mode.

Note: Mobilenet mode IP generation is required to run designs compiled to make use of Mobilenet mode. Check the information and files available on the sensAI website to ensure that you have the files for Neural Network Compiler 7.0 and to ensure that you are aware of the performance and resource utilization.

When using Mobilenet mode, there are two additional recommendations for your design and setting. First, it is recommended that the number of features (number of kernels) in both Depthwise and 1×1 Convolution is a multiple of 8. Secondly, it is recommended that you enable the collapse layer feature.

5.3.2. Compact Mode for CrossLink-NX and CertusPro-NX

When creating or modifying a project, CrossLink-NX-targeted designs and CertusPro-NX-targeted designs can enable compact mode to use a reduced-resource version of the CrossLink-NX IP and CertusPro-NX IP.

Note: The **p**erformance of compact mode is usually lower than that of optimized mode. It is recommended to use compact mode only to reduce hardware resource usage. Optimized mode generally performs better than compact mode.

5.3.3. Embedded Mode

When creating or modifying a project, CrossLink-NX and CertusPro-NX targeted designs can enable embedded mode in the Impl options window to restrict the use of external memory.

Note: One can use embedded mode only if the input and output of each layer can be stored inside internal memory when the layer is being executed.

5.4. SensAI Security Flow

SensAI supports the encryption and decryption of models. One can encrypt a model through the sensAI compiler and provide it for secure use. When an encrypted model is provided as input, sensAI will decrypt it internally, minimal information is visible, and no weights or network information can be extracted while generating firmware through sensAI. Model encryption and decryption flow are only available for the Caffe, Tensorflow, and Keras frameworks.

5.4.1. Model Encryption

Sample command to encrypt the model.

```
$ ./lsc_ml_compl --cryptography --input_file_path <input_model_path>.pb --output_file_path
<output_model_path>.elpb --password <Password> --mode encrypt
```

```
$ ./lsc_ml_compl --cryptography --input_file_path ~/model.pb \
        --output_file_path ~/model.elpb --password SomePassword123 -m encrypt
File encrypted successfully and saved at /home/user/model.elpb
```

Figure 5.18. SensAI Security Flow: Encrypt Model

Table 5.6. SensAI Security Flow: File Extension Mapping

Frame Work	Input Extension	Encrypted Extension
Keras	.h5	.elh5
Tensorflow	.pb	.elpb
Caffe	.proto	.elproto
	.caffemodel	.elcaffemodel
ONNX	.onnx	.elonnx

Note: The encrypted model can be use directly in the sensAI compiler. The compiler internally decrypts the model without exposing any weights or network details.

To use an encrypted model, select the *Encrypted* model option in the **Files of Type** section of the model selection window, as shown below.

Using an encrypted model does not alter any other steps in the compilation flow.

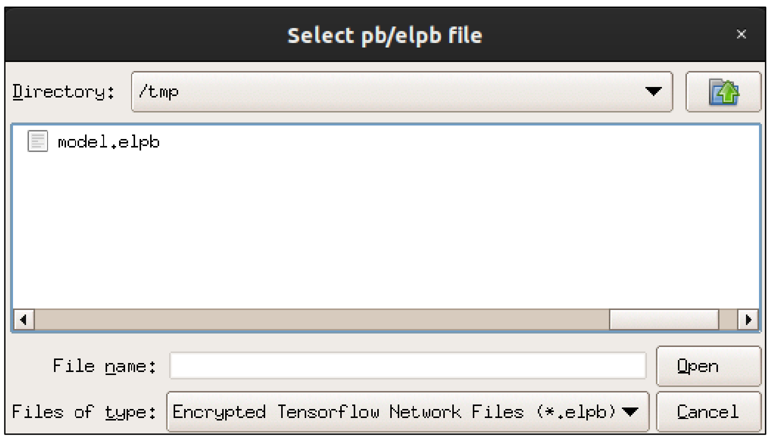


Figure 5.19. SensAI Security Flow: Encrypted Model Selection

5.4.2. Model Decryption

To decrypt the model, the user needs to have the password used during encryption.

```
$ ./lsc_ml_compl --cryptography --input_file_path ~/model.elpb --output_file_path
~/model_decrypted.pb --password SomePassword123 -m decrypt
```

```
$ ./lsc_ml_compl --cryptography --input_file_path ~/model.elpb \
        --output_file_path ~/model_decrypted.pb --password SomePassword123 -m decrypt
File decrypted Successfully and saved at /home/user/model_decrypted.pb
```

Figure 5.20. SensAI Security Flow: Encrypt Model

Note:

Without the correct password, the model cannot be decrypted.

The firmware generation from the sensAI compiler model doesn't need to be decrypted. It is for utility purposes only.

6. Supported Frameworks

Currently, the Lattice Neural Network Compiler Software supports the Caffe, TensorFlow, Keras, and ONNX (experimental) machine learning frameworks. Caffe protofiles are natively supported, while TensorFlow requires creating a frozen deployment model file.

Each supported framework is clearly defined in the appendix sections. These following sections explain how to customize or alter the neural network.

6.1. Caffe

The support for caffe has been deprecated starting from sensAI 8.0. Please use older version.

6.2. TensorFlow

The Lattice Neural Network Compiler Software can run designs created with the TensorFlow framework. It uses an internal tool to analyze and convert TensorFlow neural networks into a compatible ONNX model. You must provide a TensorFlow inference frozen model file (.pb) that includes both the graph and parameter values. This file must be optimized by removing all the nodes related to data processing or training, and all parameter variables required for inference must be converted to constants.

The frozen .pb file must contain both the network topology and constant weights for inference. Follow the instructions in the Training to Inference Conversion section to convert a training .pb model into an inference frozen .pb model.

Requirements for creating a TensorFlow inference frozen model file:

- Ignore data pre or post-processing subgraphs and operations. Use a separate script to preprocess input data so it can be used directly when testing your TensorFlow model in the Lattice Neural Network Compiler Software.
- Ensure only one placeholder exists for data input, and explicitly specify its shape in the TensorFlow standard four-dimensional image input format and dimension order.
- Do not use a frozen model from a training session or checkpoint folder. These are not supported and cannot be directly used to create a compatible project. Perform training-to-inference optimization conversion for any training model you plan to use with sensAI.
- Supported output layers include Conv2D, Matmul (for inner product and full connect), and Global Average. Activation functions such as softmax and sigmoid are supported.

Recommended guidelines (not required):

- Call tf.reset_default_graph() immediately before initializing a new inference session. Within the session, perform only inference-related TensorFlow operations. Use tf.train.write_graph to save the session graph definition as a .pb file, which can then be optimized and frozen for inference applications.
- Ignore any data pre or post-processing (for example, mean and scale) from the .pb file.Specify these in the tool or in a separate Python script layer. It is recommended to preprocess input data, save it as a raw array (.npy) file, and use this raw input array as input.
- Use Scaling and BatchNorm layers periodically to optimize performance under the fixed-point notation constraints of the hardware.
- Choose an input size that is a power of two for better computational speed and to minimize memory alignment issues.

6.2.1. Training to Inference Conversion

TensorFlow training models must be converted to inference models to be compatible with sensAI. There are three main steps in the process for converting a TensorFlow training model (located in the checkpoint directory) into the supported TensorFlow inference frozen model, which are detailed below:

1. Identify the input and output nodes needed for inference. The input node should be the node after all pre-processes, and the output node should be the node right before the post-process, normally right after the conv2D or matmul node.
2. While using TensorFlow 1.x, use tensorflow.python.tools.optimize_for_inference_lib.optimize_for_inference to remove nodes that are not related to inference, and use tf.train.write_graph to save the output in the binary .pb format.
3. Copy the output of step 2 (the simplified inference .pb) into the checkpoint folder and use tensorflow.python.tools.freeze_graph to freeze the checkpoint weight as a TensorFlow inference frozen model file (.pb).

An example graph (Cifar10 Binary NN model before and after inference optimization) is shown in Figure 6.1 and Figure 6.2.

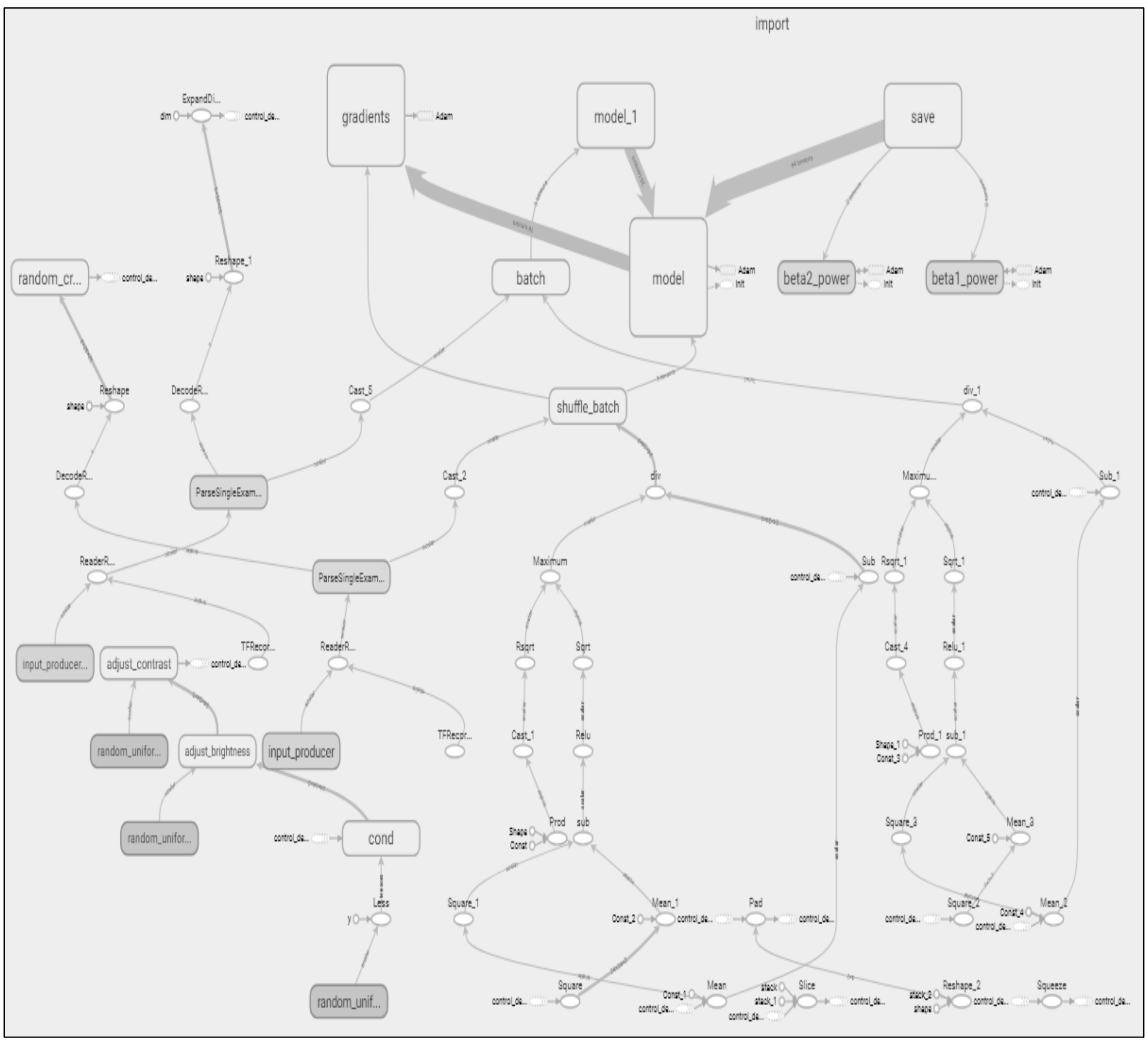


Figure 6.1. Original TensorFlow Training Model

Figure 6.1 displays an example training model. This one is not yet frozen for inference and has many extraneous nodes. These nodes are not needed for inference. Nodes that are only related to preprocessing, training, or post-processing can all be removed without affecting the precision of the inference.

After following those three steps, the same model in Figure 6.1 is optimized for Figure 6.2. It is in the form of a supported binary inference frozen model, with only inference nodes in the graph.

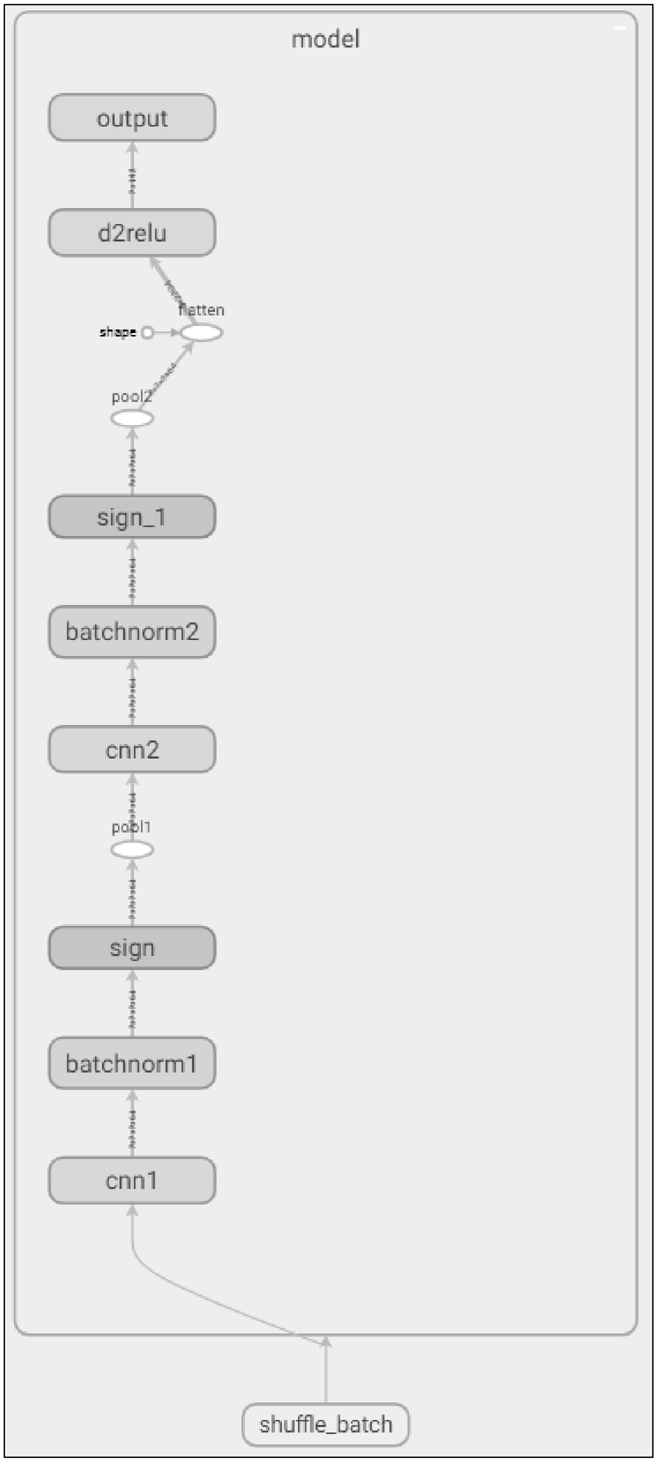


Figure 6.2. Simplified TensorFlow Inference Model

A complete standalone demo script is provided in your sensAI installation directory in "\networks\TrainToInference\checkpoint\" to demonstrate the above method. If TensorFlow and Python are already installed on your system, you can directly run trainckpt2inferencepb.py to output a frozen inference .pb file (TrainToInference.ckpt_frozenforInference.pb) for the checkpoint inside the demo. This script also supports using Docker to run on both Windows and Linux systems, allowing it to function even when Python and TensorFlow are not installed. Refer to README.txt and RUNDOCKER.txt inside the demonstration directory for more details.

There are two methods you can use to provide the input and output node information that is required for this script to run.

- Method 1: Directly provide the full name of the input and output nodes as the input parameters.
- Method 2: Use the pre-defined INPUTNODE_TAG and OUTPUTNODE_TAG as part of the node name.
  - The demo script assumes that only one input node has the "INPUTNODE_TAG" string as part of its name and that only one output node has the "OUTPUTNODE_TAG" string as part of its name. Exact input and output node names are not required as input parameters, as long as you use the following two tags pre-defined in the sensAI NN compiler:
    - INPUTNODE_TAG='_SensAI_BeginNode'
    - OUTPUTNODE_TAG='_SensAI_EndNode'

6.2.2. Binary Neural Networks (BNN)

TensorFlow does not provide an official implementation for binarization. Therefore, binarization support is experimental and limited only to three operations:

1. Sign operation
2. Conv2D
3. Matmul

Binary models created by open-source packages, for example, TensorLayer, need to have a similar computation topology to the BNN demo model. SensAI utilizes a custom implementation of Caffe for incorporating Binary Neural Networks, meaning that binary TensorFlow models must match the customized Caffe implementation.

Use this Python code to implement binarization for conv2D in TensorFlow to match the customized Caffe implementation:

Python Binarization Implementation

```
              tf.multiply(tf.sign(x),tf.reduce_sum(
tf.abs(x),[0,1,2])/ tf.to_float(tf.size(x)/x.get_shape().as_list()[3]  )  )
```

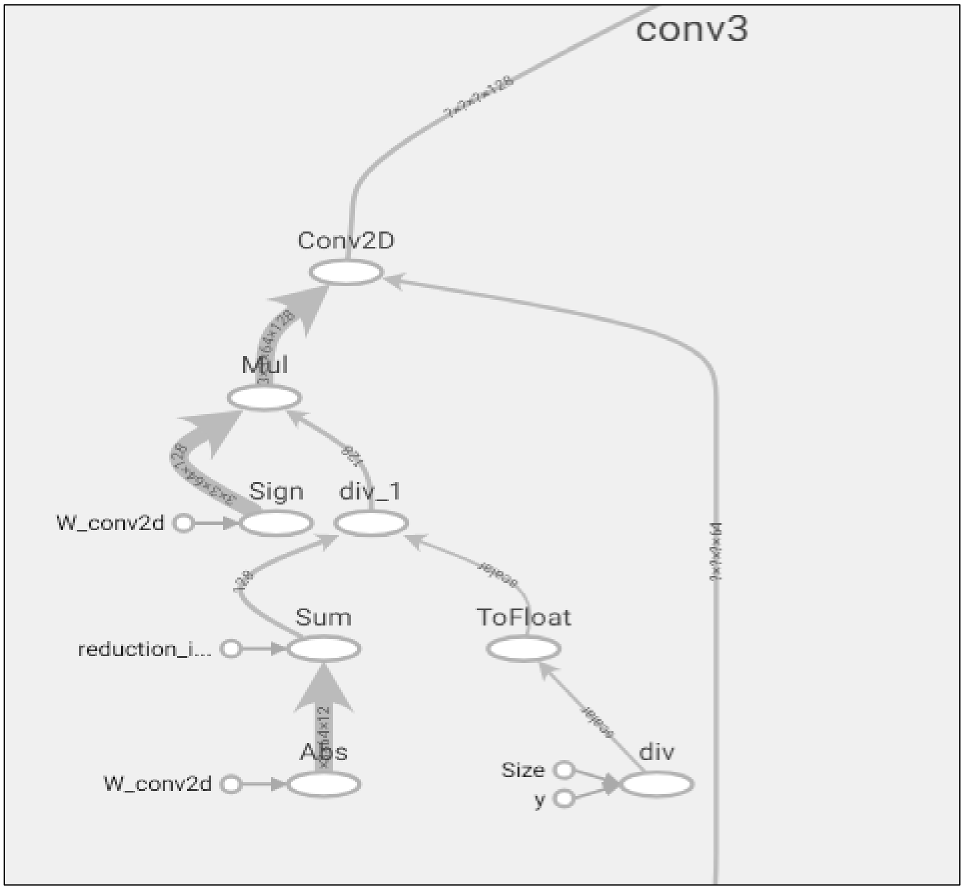


Figure 6.3. Tensorboard Visualization of Binarization

The Python code and TensorBoard representations may be difficult to understand. The following C++ code (inside customized Caffe) to implement the above computation topology is equivalent. It demonstrates how the binarization algorithm works.

Implementation C++ Code

```
#define sign(x) ((x)>=0?1:-1)
  const int div = weights->count() / weights->num();
  for (int num = 0; num < weights->num(); num++) {
    if (normalized_weights) {
      A[num] = 0;
      for (int _c = 0; _c < weights->channels(); _c++)
      for (int _h = 0; _h < weights->height(); _h++)
      for (int _w = 0; _w < weights->width(); _w++)
        A[num] += std::abs(weights->data_at(num, _c, _h, _w)) / Dtype(div);
    }
  }
  for (int index = 0; index < weights->count(); index++) {
    const int num = index / div;
    wb->mutable_cpu_data()[index] = A[num] * sign(weights->cpu_data()[index]);
  }
```

In addition, due to the limitations of the hardware and precision of fixed-point representation, you must follow these requirements when creating a binary TensorFlow inference frozen model:

- When using signed operations in a binary TensorFlow model, bear in mind that the hardware only supports either 0/1 or -1/1 quantization modes. Additional preprocessing must be implemented so that the subgraph can generate 0/1 or -1/1 as the output and produce the expected results in hardware. The constant "y" is equal to 0.5.

0/1 Mode (UltraPlus) –1/1 Mode (ECP5)

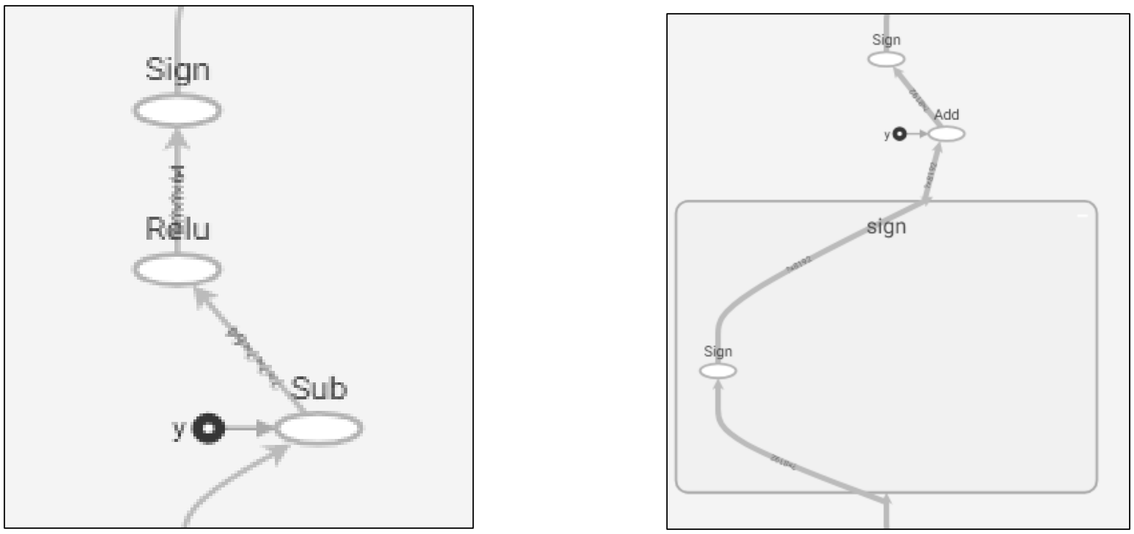


Figure 6.4. Binary Neural Network Modes in TensorFlow

- A batch normalization operation is required right after conv2D operations (with binarized normalization).
- Currently, NNC does not support a mixed model. In binary TensorFlow models, all conv2D operations and all Matmul (full connect layer) need to be binarized (sign operations similar to Keras Sample Code below need to be part of weight loading). If a model is not a binary model, then the sign operation should not be present in the graph at all.

6.3. Keras

NNC supports implementing Keras networks in the form of "tensorflow.keras" designs shipped with TensorFlow 1.14, 2.0, 2.3, 2.5, and 2.9. The Keras/Keras-Team release version of Keras is unsupported. The slight implementation differences likely result in your design not being compatible with sensAI, if your model is created with the Keras team release instead of the TensorFlow release.

NNC requires a single HDF5 file (.h5 with both weight and architecture) for Keras models. It is recommended to set Keras to inference (tf.keras.backend.set_learning_phase(0)) before saving it as a .h5 file, as NNC only supports inference model format. If the .h5 is saved as a training format file, NNC attempts to convert it to inference. But it is not guaranteed that this converted Keras model can produce the same output as the original Keras model.

NNC uses the channel_first data format for intermediate graph representation. Simulation output as well as the engine provided output in NNC will be in channel_first format only. For raw numpy sample input, the user needs to provide channel_last formatted data for Tensorflow and Keras. In addition to these file requirements, Keras models are also subject to the same hardware limitations and parameter constraints as supported by TensorFlow layers.

6.3.1. Using Keras

As an example of how to use Keras, the humanGesture design can be implemented in Keras using the following code. In some cases, it is required (for example, using the Lambda function for 8-bit quantization for Lattice NNC) that the user convert the Keras model (.h5) to the Tensorflow model (.pb) to avoid any bad marshal data type errors. To help convert the Keras model to TensorFlow format, please use the reference script at networks\TrainToInference\keras2tf_conversion\keras2tf.py.

Keras Sample Code

```
def humanGesture(input_tensor, classNumer=4,epsilonBN=1e-3 ):
       a = Input( tensor=input_tensor)
       x=Conv2D(24, (3, 3) ,padding='same')(a)
       x=BatchNormalization(epsilon=epsilonBN)(x)
       x=Activation('relu')(x)
       x=MaxPooling2D(pool_size=(2, 2))(x)
       x=Conv2D(20, (3, 3) ,padding='same')(x)
       x=BatchNormalization(epsilon=epsilonBN)(x)
       x=Activation('relu')(x)          #Fire 3
       x=Conv2D(20, (3, 3),padding='same' )(x)
       x=BatchNormalization(epsilon=epsilonBN)(x)
       x=Activation('relu')(x)
       x=MaxPooling2D(pool_size=(2, 2))(x)
       x=Conv2D(22, (3, 3),padding='same' )(x)
       x=BatchNormalization(epsilon=epsilonBN)(x)
       x=Activation('relu')(x)           #Fire 5
       x=Conv2D(22, (3, 3),padding='same')(x)
       x=BatchNormalization(epsilon=epsilonBN)(x)
       x=Activation('relu')(x)
       x=MaxPooling2D(pool_size=(2, 2))(x)
       x=Conv2D(24, (3, 3),padding='same')(x)
       x=BatchNormalization(epsilon=epsilonBN)(x)
       x=Activation('relu')(x)
       x=MaxPooling2D(pool_size=(2, 2))(x)
       x=Flatten()(x)
       x=Dense(classNumer , kernel_initializer='uniform' )  (x)
       model = Model(inputs=a, outputs=x)
```

6.3.2. Using ONNX

NNC requires an ONNX file (.onnx) for ONNX models. The model can be a float or PTQ model. While loading the model, the create_quantized_version option needs to be selected. This section shows how to convert the model trained in PyTorch to ONNX which can then be loaded in the NNC. As support is experimental, you may find that some layers or attributes are not supported by NNC for the converted ONNX model.
The following is the code to convert the pytorch mnist model to ONNX using the *torch.onnx.export* function.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.onnx import register_custom_op_symbolic
from torch.autograd import Function

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),
(0.5,))])

# Load the training and test datasets
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Function to calculate the padding for "same" convolution
def calc_pad(kernel_size, stride, dilation=1):
    padding = ((stride - 1) + dilation * (kernel_size - 1)) // 2
    return padding

class quant_node(nn.Module):
    def __init__(self, constant=0.2):
        super(quant_node, self).__init__()
        self.constant = constant

    def forward(self, x):
        return x * self.constant
# Register the custom op for ONNX export
def multiply_by_constant_symbolic(g, x, constant):
    return g.op("quant", x, torch.tensor(constant, dtype=torch.float32))

# Ensure that the custom op is registered with the appropriate name and version
register_custom_op_symbolic("::quant_node", multiply_by_constant_symbolic, 13)
```

```
class MyReLUFunction(Function):

    @staticmethod
    def symbolic(g, input):
        return g.op('custom', input)

    @staticmethod
    def forward(ctx, input):
        ctx.input = ctx
        return input.clamp(0)

    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()
        return grad_input

class MyReLU(nn.Module):
    def forward(self, input):
        return MyReLUFunction.apply(input)

# Define the neural network model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.relu = nn.ReLU()

        #cbsr1
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3,stride=1,
padding=calc_pad(3,1),bias=False)
        self.bn1 = nn.BatchNorm2d(16, momentum=0.9, eps=0.001 )

        #cbsr1
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3,stride=1,
padding=calc_pad(3,1),bias=False)
        self.bn2 = nn.BatchNorm2d(16, momentum=0.9, eps=0.001 )

        #dw
        self.dw_1 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=calc_pad(3,1),
groups=16, bias=False)
        self.pt_1 = nn.Conv2d(16, 16, kernel_size=1,stride=1, bias=False)

        # self.bn2 = nn.BatchNorm2d(16, momentum=0.9, eps=0.001 )

        self.conv3 = nn.Conv2d(16, 16, kernel_size=3,stride=1,
padding=calc_pad(3,1),bias=False)
        self.bn3 = nn.BatchNorm2d(16, momentum=0.9, eps=0.001 )
        self.dp = nn.Dropout2d(p=0.2)
        self.nnfl1 = nn.Flatten()
        self.fc1 = nn.Linear(3136, 10)
```

```
        self.quant = quant_node(0.2)
        self.cus_relu = MyReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x1 = self.pool(x)
        # # dw conv
        x = self.dw_1(x1)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.pt_1(x)
        x = self.bn2(x)
        x2 = self.relu(x)
        x = torch.add(x1, x2)
        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu(x)
        x = nn.functional.dropout(x)
        x = x.view(-1, 3136)
        x = self.fc1(x)
        return x

# Instantiate the model, define the loss function and the optimizer
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop
num_epochs = 1
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 100 == 99:
            print(f"[{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f}")
            running_loss = 0.0

print('Finished Training')
```

```
# Save the trained model
torch.save(model.state_dict(), 'mnist_cnn.pth')

# Export the model to ONNX
dummy_input = torch.randn(1, 1, 28, 28)  # MNIST images are 1x28x28
torch.onnx.export(model, dummy_input, "toy_mnist_3.onnx", opset_version=13,
input_names=['input'], output_names=['output'],
                  dynamic_axes={'input': {0: 'batch_size'}, 'output': {0: 'batch_size'}}
                  , export_params=True, training=torch.onnx.TrainingMode.EVAL,
do_constant_folding=False,)

print('Model has been exported to ONNX')
```

7. SB Debugging

The USB debugging feature in NNC allows you to debug iCE40, ECP5 (using the USB3-GbE VIP IO Board), CrossLink-NX, and CertusPro-NX designs. The DRAM and registers of the ECP5 device can also be accessed using this option.

7.1. Hardware Configuration

The following steps are required to configure the hardware before using it for USB debugging in the sensAI tool.

7.1.1. ECP5

1. Refer to the USB3-Gigabit Ethernet Demo User Guide (FPGA-UG-02054).
2. Configure the FX3 USB controller.
    - Follow Appendix B in the user guide document.
    - Select the image file mentioned in step 5 from the following location:
      utils\drivers\lattice-usb\cyfxuvc.img
3. Configure ECP5.
    - Follow the ***ECP5 SPI Flash Programming*** section in the USB3-Gigabit Ethernet Demo User Guide (FPGA-UG-02054) document.
4. Select the debugging bit file.
    - For all designs, select the bit file from the following location:
      utils\drivers\lattice-usb\bitfiles.zip
      Refer: utils\drivers\lattice-usb\README
      Note that as there is no DRAM on UltraPlus, USB debugging must be done using ECP5/CNX/CPNX hardware, and DRAM can be interfaced to see the input and output blob data only.

7.1.2. CNX VVML, CPNX

1. Flashing the FX3 USB .img file.
    - Connect the jumper to port **J13** of the Crosslink-NX or CPNX VVML Board (Rev B) and connect the board to the PC using a USB3 cable.
    - Connect the jumper to port **J4** of the Avant board and connect the board to the PC using a USB B-mini cable.
    - Open the USB control center application (the Cypress FX3 SDK needs to be installed for the same).
    - Press the push-button switch ***SW2*** on the board to reset the FX3 chip.
    - You can see the bootloader device, as shown in Figure 7.1.

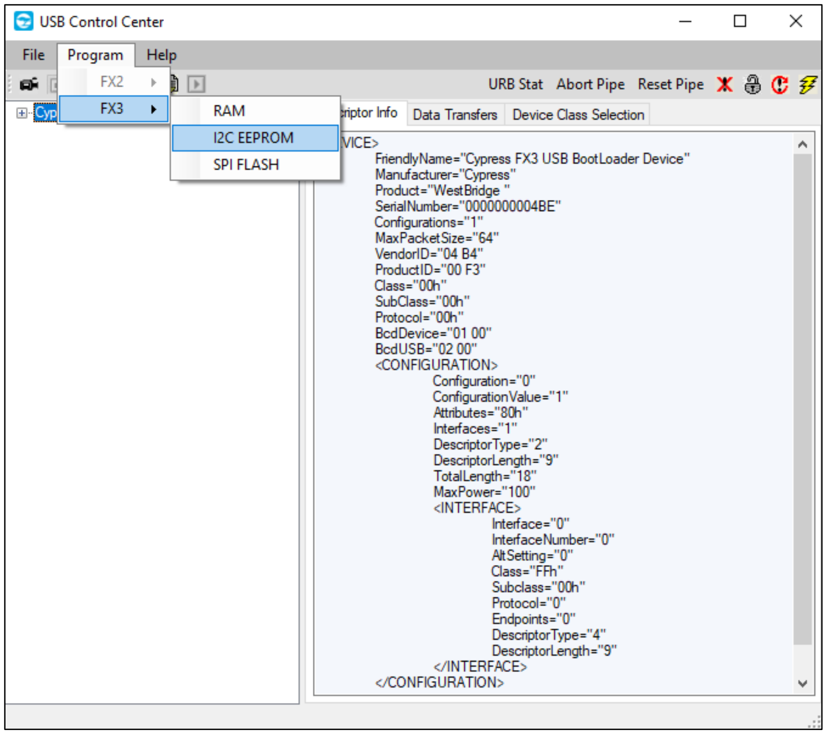


Figure 7.1. Cypress Window

- Select the Cypress USB Bootloader.
- Select ***Program > FX3 > I2C EEPROM*** from the menu bar.
- Browse and select the USB debug file LSCVVML.img from the path *utils\drivers\lattice-usb*.
- Wait until ***Programming of I2C EEPROM Succeeded*** appears in the taskbar at the bottom of the window.
- Remove the jumper from port ***J13***.
- Power off and power on the board. FX3 should boot from the I2C E2PROM.

2. Erasing the CNX VVML and CPNX prior to reprogramming.

   If the CrossLink-NX Voice and Advanced device is already programmed, either directly or loaded from SPI Flash, follow the given procedure to first erase the CrossLink-NX Voice and Advanced SRAM memory before re-programming the CrossLink-NX-Voice and Advanced SPI Flash. While doing this, keep the board powered ON when re-programming the SPI Flash so that it does not reload on reboot.

   Note: Before erasing, disconnect the ***J13*** jumper.

- Launch the Lattice Radiant Programmer. Create a new blank project.

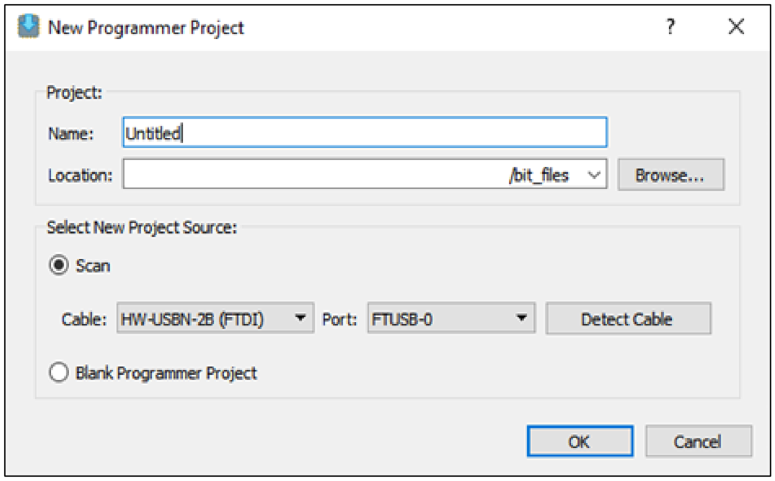


Figure 7.2. Radiant Programmer – Default Screen

- Select **LIFCL** for **Device Family** and LIFCL-40 for Crosslink-NX. Then select LFCNX for the CertusPro-NX device, as shown in Figure 7.3.

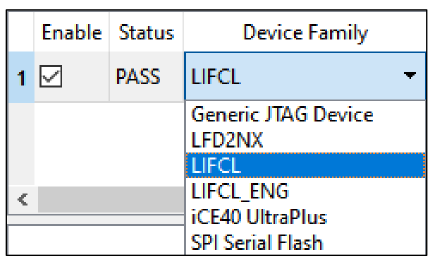


Figure 7.3. Radiant Programmer Device Selection

- Right-click and select **Device Properties.**
- Select **JTAG** for **Port Interface**, **Direct Programming** for **Access Mode,** and **Erase Only** for **Operation** as shown in Figure 7.4**.**

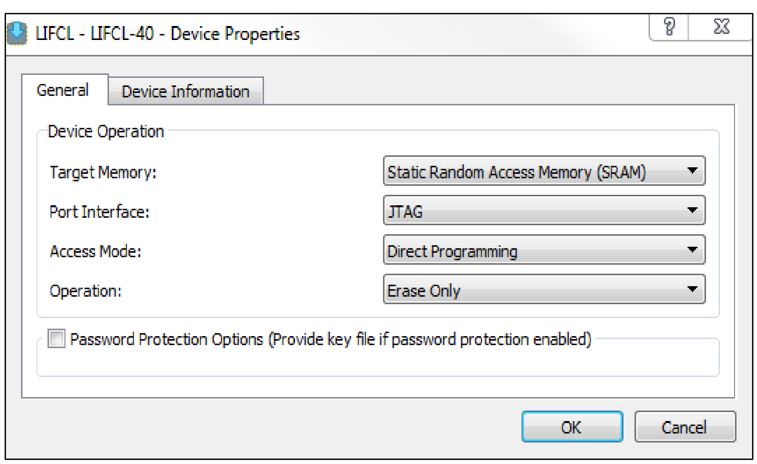


Figure 7.4. Radiant Programmer – Device Operation

- Click **OK** to close the Device Properties dialog box.
- Now press the **SW5** push-button switch on the board before clicking the program button as given in the next step, and keep it pressed till you see the **Operation Successful** message in the Lattice Radiant Programmer log window.
- In the Lattice Radiant Programmer main interface, click the **Program** button to start the erase operation while keeping **SW5** pressed.

3. Programming Crosslink-NX VVML or CPNX board

All the bit files are included in the file at path **utils\drivers\lattice-usb\bitfiles.zip**. Unzip the file to select the bit file, as given in step 4 below. Also, please refer to readme for reference while selecting the bitfile. Before SPI flashing, disconnect the ***J13*** jumper that you connected while flashing the .img file.

- Ensure that the CrossLink-NX Voice and Advanced Device SRAM is erased by performing the steps given in the above section.
- In the Lattice Radiant Programmer main interface, right-click on Operation and select **Device Properties** to open the Device Properties dialog boxes, as shown in Figure 7.5.

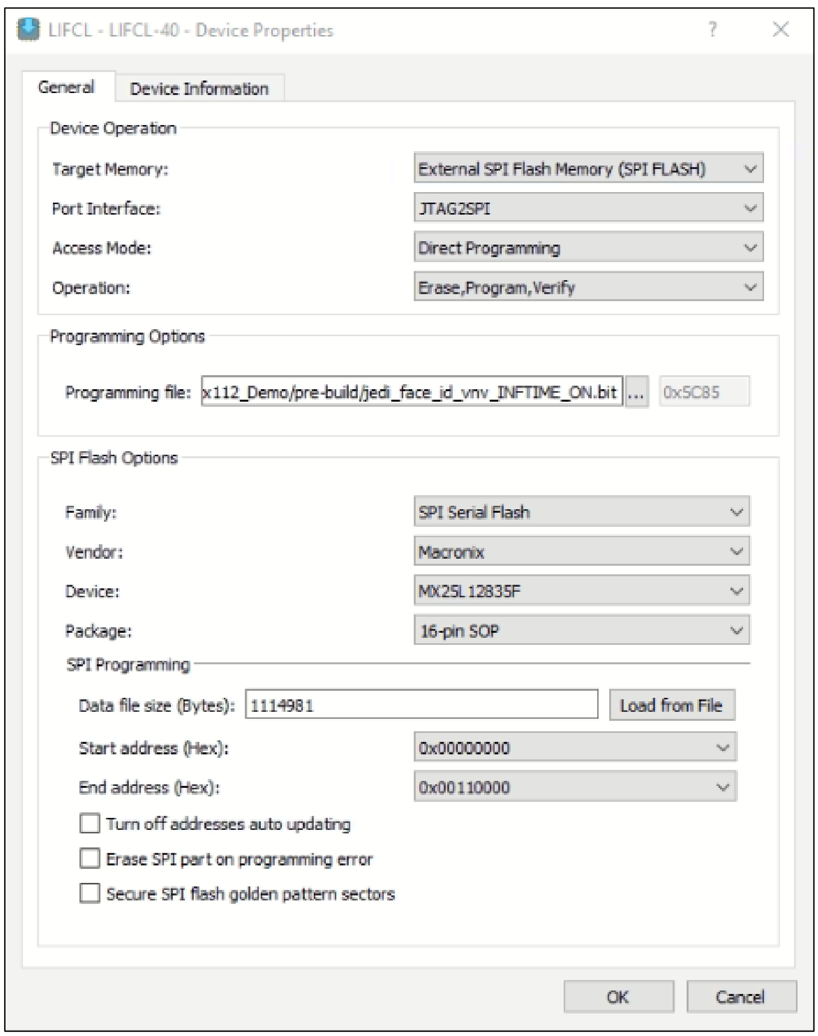


Figure 7.5. Selecting Device Properties for CrossLink-NX

- Select SPI Flash for Target Memory, JTAG2SPI for Port Interface, and Direct Programming for Access Mode.
- Select the bit file you want to flash by extracting the zip file given at the path: utils\drivers\lattice-usb\bitfiles.zip and selecting the bit file from there.
- For SPI Flash Options, make the selections in Figure 7.5 given above and select **Macronix 25L12833F** as the device.
- Click Load from File to update the data file size (bytes) value.
- Ensure that the following addresses are correct.
    - Start Address (Hex): **0x00000000**
    - End Address (Hex): **(Start Address + size of bit file)**
- Click **OK**.
- On board, press the **SW5** push button switch before clicking the program button in the step below and keeping it pressed till the **Operation Successful** message is seen in the Lattice Radiant Programmer log window as shown in Figure 7.6.
- From the Lattice Radiant Programmer main interface, click the **Program** button to start the programming operation.

Figure 7.6. Output Console after Successful Flashing

7.1.3. Avant Device

For USB debugging on an Avant device, you will need a Cypress USB FX3 board. Connect the Avant board and USB FX3 board as shown in Figure 7.7.

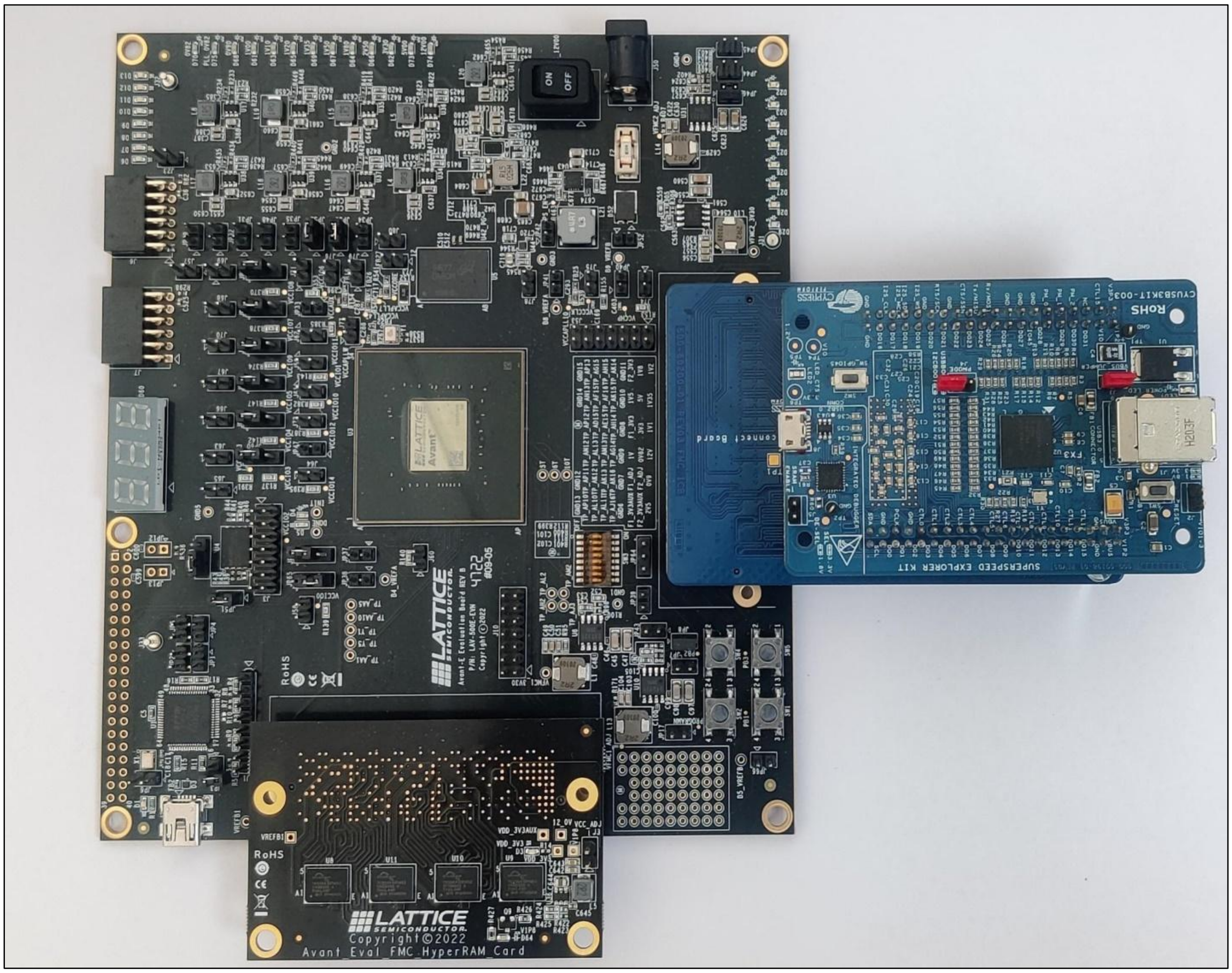


Figure 7.7 Avant Board with FX3 USB Board

- Upload the LSCVVML.img file to the Cypress FX3 USB board, keeping the jumper configuration as:
  - Jumper J4 being open.
  - Jumper J3 shorted.
- Upload the bitfile of Advanced IP to the board using the Lattice Radiant Programmer.
  - Using a USB port for the Avant board for uploading a bitfile to the FPGA.
- Use the FX3 port for reading HW values from the board.

7.2. Debug Window Options

To launch the USB debugging window from the SensAI GUI, click on **Tools > USB Debugging** from the main window. The USB debugging window (Figure 7.8) opens.

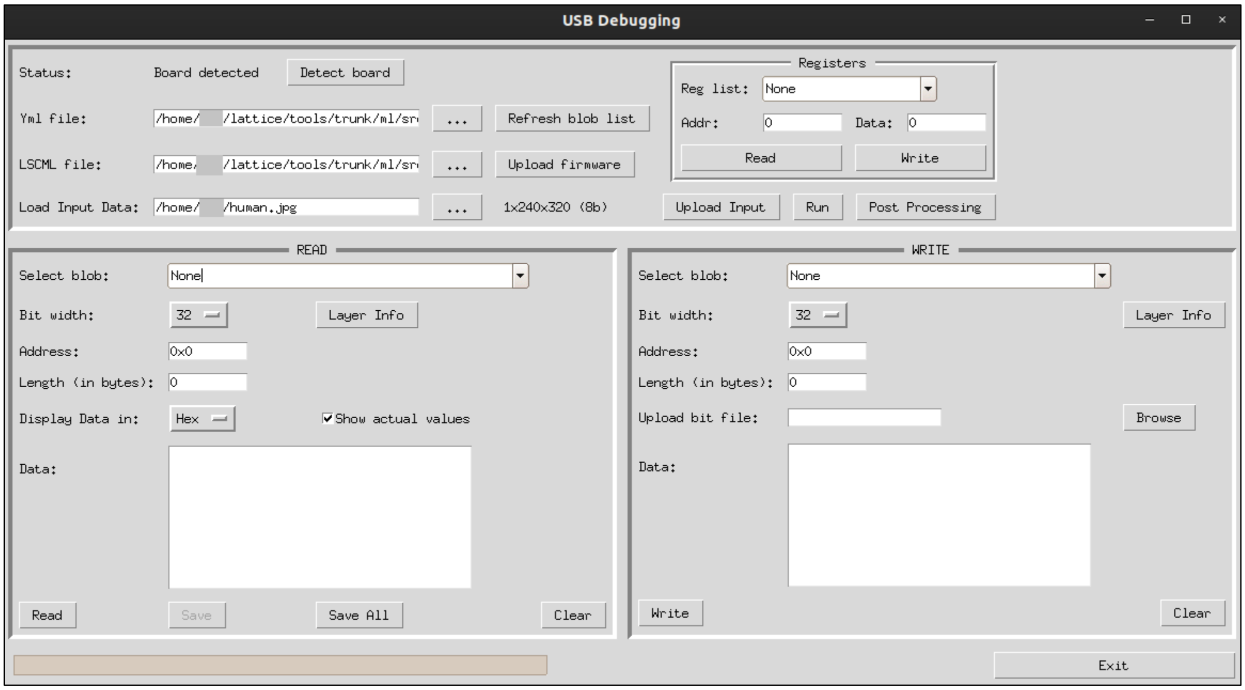


Figure 7.8. USB Debug Window

- **Status:** Indicates if the board is detected. Read and Write operation buttons are disabled until the board is detected by the software.
- **Detect Board:** Click this to retry connecting to the board.
- **Yml File:** Provide a YML file to parse the blob layer name, Q-format, and starting address. After reading the YML file, ***Select blob*** displays available blob names, ***Address*** shows the starting address of the selected blob, ***Length*** shows the total size, and ***Bit width*** displays the bit width of data to read or write.
- **Refresh Blob List:** Refreshes the blob list. Use this if the YML is changed while the debugger is running.
- **LSCML File:** The .lscml file path generated by the tool needs to be uploaded on board as firmware. The file is automatically detected if the current project already has an associated .lscml file.
- **Upload Firmware:** Upload the firmware file to the board. This functionality is disabled until the board is detected.
- **Load Input Data:** Image or raw input file to load at the input blob. Accepts .jpg, .png and .npy format.
- **Upload Input:** Based on the resolution selected in the drop-down menu, image data is pre-processed and uploaded to the input blob address on board. Disable it until the board is detected. A valid YML file is required for this operation.
- **Reg List:** Drop-down option for all the register lists. Below is the table for all the registers with their address information.
- **Registers Read/Write:** Register read and write operations to and from addresses mentioned in the address box. Disable it until the board is detected. Addr and Data box values are in hexadecimal for read and write operations. More details on registers can be found in Appendix D. USB Debugging Register Map.
- **Run:** This operation runs the engine once. All the blobs are updated based on input image data.
- **Post Processing**: This option is enabled only when the USB debugging window is launched from an opened project. If the post processing command is configured in the project settings as shown in Figure 3.2, then this operation runs the post processing script on input data (a selected image or .npy) with the last blob .npy file.

- **Select Blob:** Select a blob (by name) as the target of your read and write operations. Blob names are displayed based on the YML file.
- **Layer Info:** This button is enabled only when the USB debugging window is launched from an opened project. After selecting this button, a window with information about that blob is launched. This information includes the blob dimension, memblks, height_per_mem/depth_per_mem, DRAM address, output EBR list, and a table that shows the details on how values are divided into memblks/EBRs.
- **Address:** The starting address of DRAM. This is shown after selecting a blob name. The Blob address is based on a YML file. This DRAM address can be changed.
- **Length:** Total size of data to read or write. This is shown once a blob name is selected. The total blob length is based on the YML file. The length can be changed.
- **Display Data In:** Selects the format in which data should be read, either hexadecimal or floating point. Hex is the default setting. Selecting Float converts received data into a floating point using the selected blob layer Q-format.
- **Show Actual Values:** This checkbox is enabled only when the USB debugging window is launched from an opened project. Enabling this checkbox filters out extra values that are read from the memblks of external DRAM and displays only the actual values of the blob.
- **Upload Bit File:** Writes data in hex into a DRAM address. This option is only necessary when you wish to perform a write operation.
- **Data:** Displays the read operation data either in hex or float, and uploads bit file data in hex.
- **Read:** Performs a read operation.
- **Write:** Performs a write operation.
- **Clear:** Clears the data box.
- **Save:** Saves the displayed data in a file. Valid only for read operations.
- **Save All:** Saves all the blob data.
- **Exit:** Exits the debugging window.

7.3. Driver Installation

Due to requiring a USB driver to operate, your computer may not support USB debugging without first installing the device driver. This section covers the process for installing the required device driver in order to enable USB debugging.

7.3.1. Windows Driver

The driver for Windows is installed by running the lscvip.inf provided in the driver/pre-build folder of your sensAI installation. This can be done by right-clicking the file and selecting ***Install***. To manually install the driver by selecting your USB device in Device Manager and selecting ***Update Driver***, you need to navigate to the driver/pre-build directory and select the “lscvipdrv.dll” file.

Driver Signature Enforcement needs to be disabled to install this driver. If you encounter an error related to the driver signature, the following steps guide you through the process of disabling this temporarily for installation.

Driver Signature Enforcement Settings for Windows

1. Get to the advanced boot options menu. You can hold down the Shift key while you click the “Restart” option in Windows 8 or 10. Your computer thus restarts into the advanced boot menu.
2. Select the **Troubleshoot** tile on the **Choose an Option** screen that appears.
3. Select **Advanced Options**.
4. Click on **Startup Settings** tile.
5. Click the **Restart** button to restart your PC on the Startup Settings screen.
6. Select the **Disable driver signature enforcement** option at the Startup Settings screen.
7. Your PC boots with driver signature enforcement disabled, and you can install unsigned drivers.
8. The next time you restart your computer, driver signature enforcement can be enabled again. You need to go through this menu again to disable it if you wish to reinstall the driver for any reason.

7.3.2. Linux Driver

For Linux systems, the libusb package needs to be installed. Use the following command in your terminal to install the libusb package on Ubuntu.

```
sudo apt-get install libusb-1.0-0
```

To avoid requiring super-user permission for USB debugging, each time you wish to run the software, the device entry in your system udev rules needs to be added. Add the following line to your udev rule file, which is typically found at /etc/udev/rules.d/<file-name>.rules. Restart your udev subsystem.

```
SUBSYSTEM==”usb”, ATTRS{idVendor}==”1134”, ATTRS{idProduct}==”aa01”, MODE=”0666”
```

To restart your udev subsystem, use the following command in the terminal.

```
sudo /etc/init.d/udev restart
```

7.4. USB Debugging API Interface

SensAI allows you to perform USB debugging through an API interface in the command line, which supports the same features as the GUI and requires the same driver as detailed in the previous section. An example Python file, *‘example_usb_debugging.py’*, is provided in the sensAI installation directory to demonstrate the usage of the API interface for USB debugging.

Note that for Linux systems, using the tools via the command line without super-user permission, your driver must be installed along with making the udev changes detailed in the previous section.

7.4.1. Class Overview

To use the API interface, the usb_api class needs to be imported from usb.lib.usb_api using the command:

```
from usb.lib.usb_api import usb_api
```

The following methods are provided by the usb_api class:

- load_dll()
  - Loads platform specific USB library dll/so for interfacing with ECP5 device. This method needs to be called before any further operations.
  - Returns 1 on success and 0 on failure.
- usbInit()
  - Detects the ECP5 device over USB interface and initializes if device is found.
  - Returns 1 on success and 0 on failure.

- usbDeinit()
  - Releases the USB device. Only applicable on Linux machines.
- writeDram(address, length, bit_width, rData)
  - Writes data to the DRAM using the four required arguments.
- address
  - Base address of the DRAM where the data is to be written.
- length
  - The length of the rData specified in bytes.
- bit_width
  - The bit width of the list elements of the rData. Data is written to the DRAM as per the bit width.
- rData
  - The list of data to write.
- readDram(address, length, bit_width, sData)
  - Reads data from the DRAM. Following is the argument description:
    - address: Base address of the DRAM where the data is to be read.
    - length: The length of the sData, which is specified in bytes.
    - bit_width: The bit width of the list of elements of the sData. Data is read from the DRAM as per the bit width.
    - sData: The container for the data that is to be read.
- regRead(address)
  - Reads the register value of the register specified by *address* and returns it. Prints an error message in case of a failure.
- regWrite(address, data)
  - Writes the *data* to register specified by *address*.
- upload_firmware(lscml_file)
  - Reads a sensAI program (.lscml) file specified by *lscml_file* and uploads the firmware to the 0x0 address of the DRAM.
    The .lscml file is generated by sensAI during the compile stage. You must use the path to a valid .lscml file as the argument.
- upload_input(yml_file, input_image)
  - Reads the mean, scale, and fraction of the input layer from the yml file and performs preprocessing based on it. Then it uploads preprocessed data to 0x0f000000 + <input-layer-extmem-address> in DRAM.
  - The arguments, *input_image* and *yml_file,* must be paths to valid .yml and input image files, respectively. The .yml file is generated in sensAI during the Analyze stage.
- run_engine()
  - This method writes registers to trigger the CNN IP to run once. Upon completion of a single run, output is generated at 0x0f000000 + <output-blob-extmem> in DRAM. Before running this step, the firmware and the input image should be uploaded to DRAM.
  - To save the output blob data from DRAM into a file on your computer, refer to the example steps provided in the example_usb_debugging.py file in your sensAI installation directory.

7.5. Board Detection Troubleshooting

If the board does not show up, try the following steps for troubleshooting your setup to attempt to resolve the issue:

1. Check the Board.
    - If using ECP5 for debugging, check that ***USB3-GbE VIP IO Board*** is written on the bottom layer of the EVDK (Figure 7.9).

Figure 7.9. USB3-GigE VIP Board Label

- If using Crosslink-NX Voice and Advanced Board, check that ***LIFCL-VVML-BRD*** is written on the board.

Figure 7.10. CNX-VnV Board Label

- If using Certus Pro-NX Voice and Advanced Board, check that ***LFCPNX-VVML-EVN*** is written on the board.

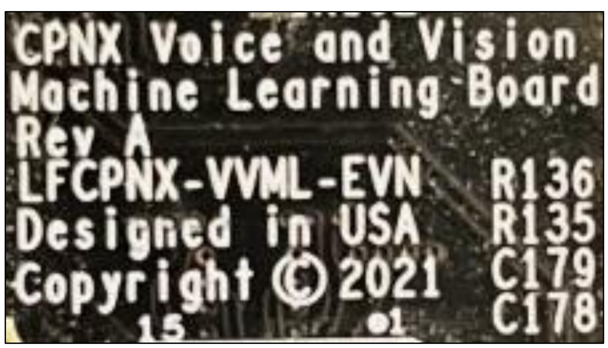


Figure 7.11. CPNX-VnV Board Label

2. Verify that you have installed the Cypress file into the Cypress chip and repositioned the jumper pins into the correct configuration.
3. For ECP5/CNX and CPNX devices, check that you have the correct bitstream programmed to the SPI Flash.
4. Ensure that the Micro USB 3.0 (not USB Mini) connector is connected from the bottom board and not the middle board.
5. For ECP5, after connecting the USB from the EVDK to the computer, press the **sys_rst** button on the top board.
6. Under Device Manager, you should now be able to see the board.

   If you still do not see the device and your computer is using Windows, you may need to disable the Windows driver certification to make it show up.

7.6. CrossLink-NX, CertusPro-NX and Avant Layer by Layer USB Debug

To debug USB values layer by layer, you can see all the layers in the blob list in the USB debugging window, as shown in Figure 7.12.

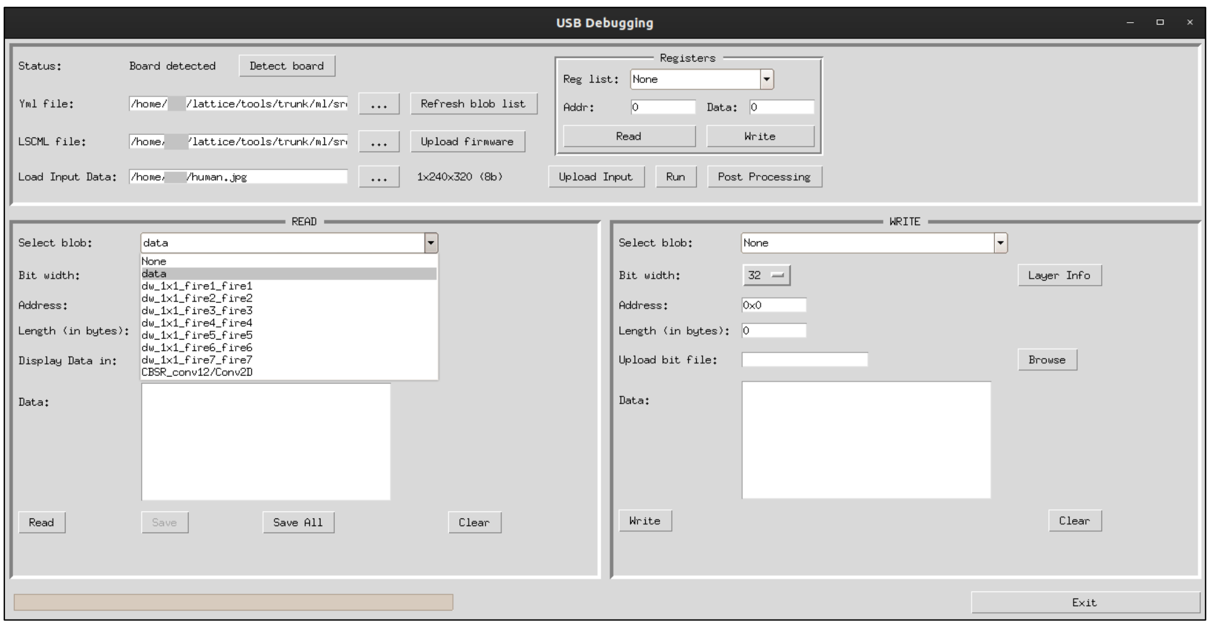


Figure 7.12. USB Debug Window

You can select one of the blobs to run USB debugging. Once you select any bob, sensAI generates USB debug firmware for the selected layer.

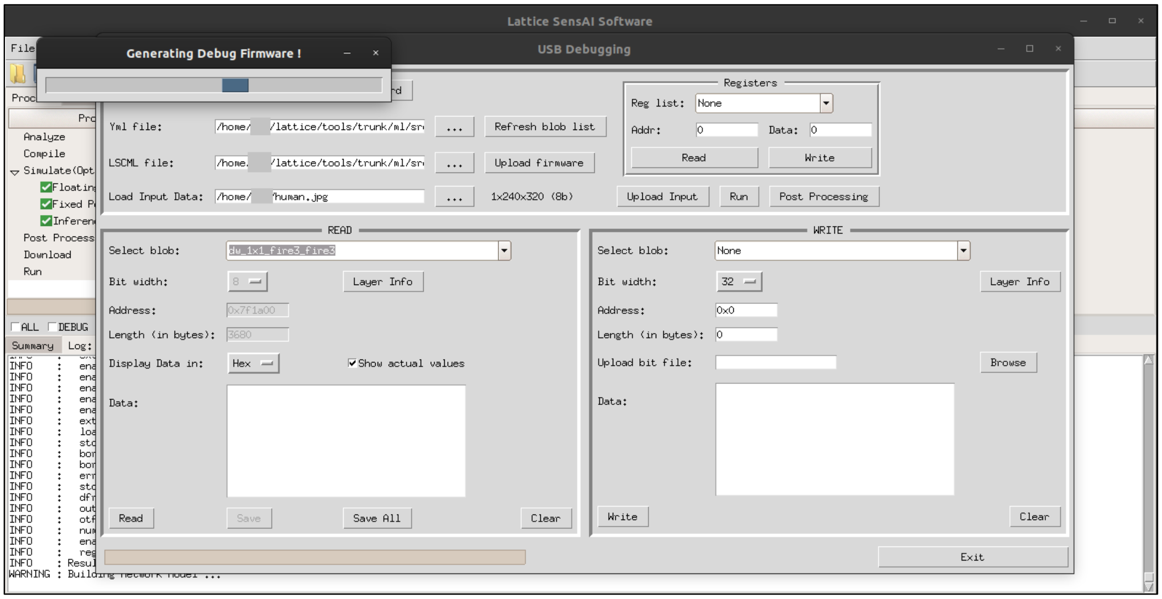


Figure 7.13. USB Debug Firmware Generation

The USB debug window sets the USB debug firmware, bit width, address, and data length based on the blob configuration.

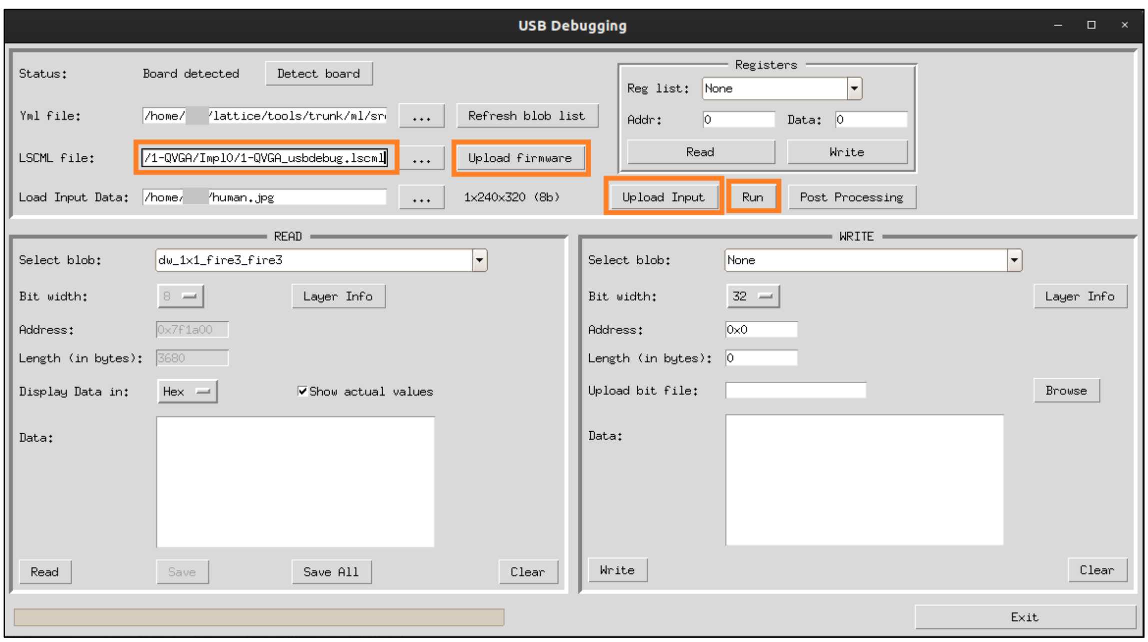


Figure 7.14. Upload FW, Input and Run USB-Debugging

Now you can:

- Upload Firmware
- Upload Input
- Run
  - To read data in the desired data type, you can select the datatype in Float or Hex.

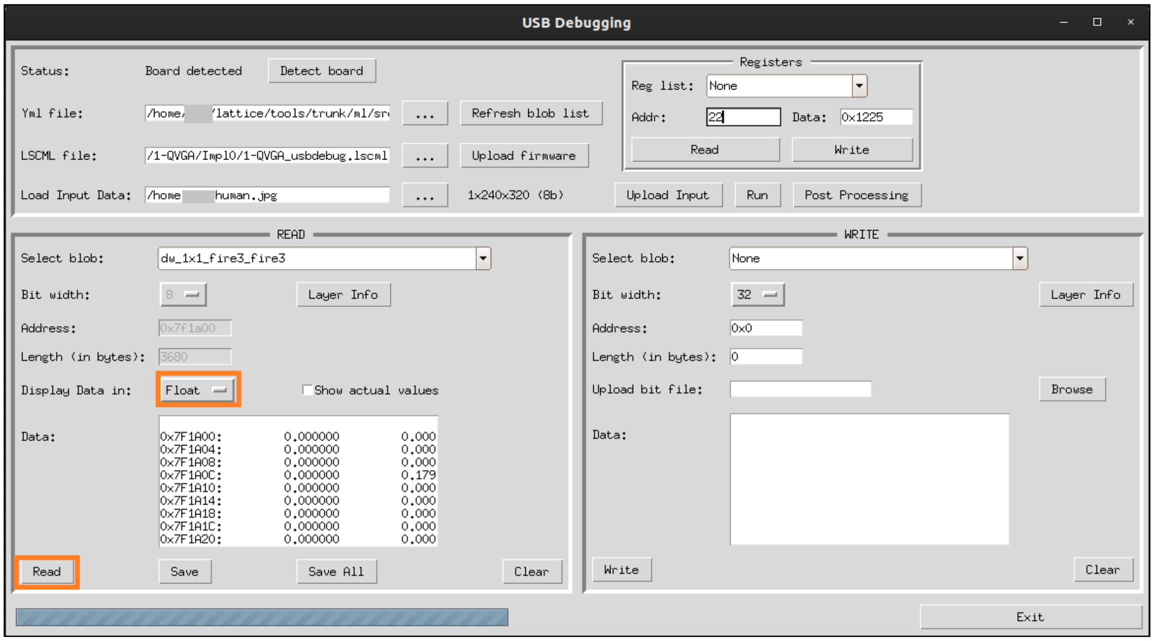


Figure 7.15. Read USB Data with Blob Selected

Notes:

- To read data from a specific address, you must select **None** in the blob list, and pass the address along with the length, and then read the data.
- On the new input data, you need to perform all the steps by first selecting the new input data and then performing all the steps.

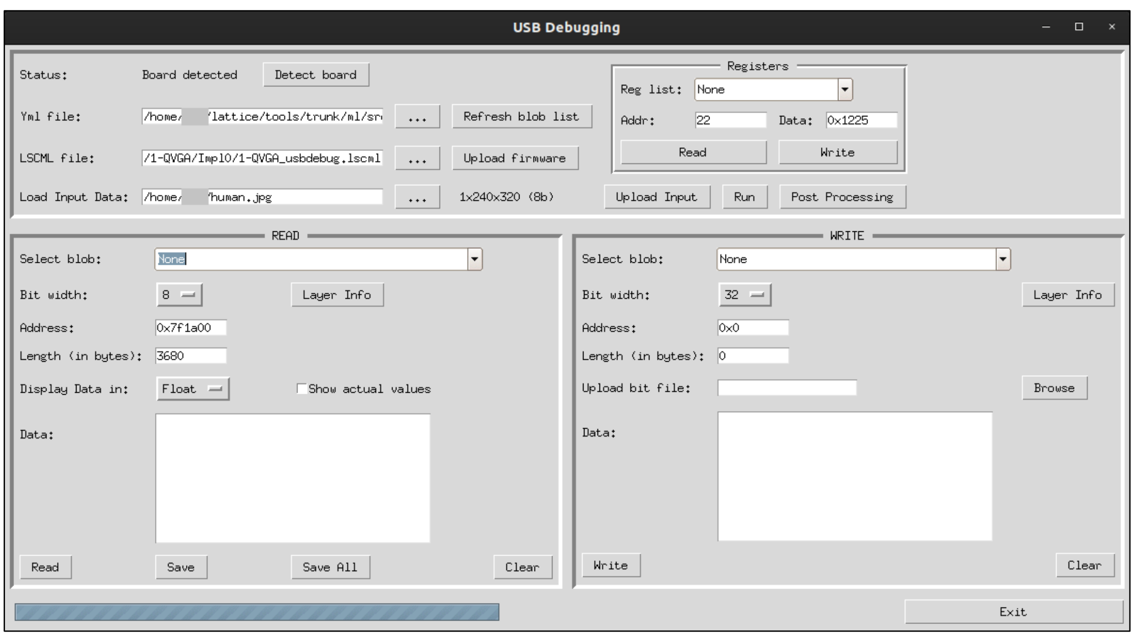


Figure 7.16. Read USB Data without Blob Selected

To save data, click **Save**. The save file dialog pops up. After saving the text file, sensAI Compiler finds the expected vs. USB Debug values MAE and shows them in a popup.

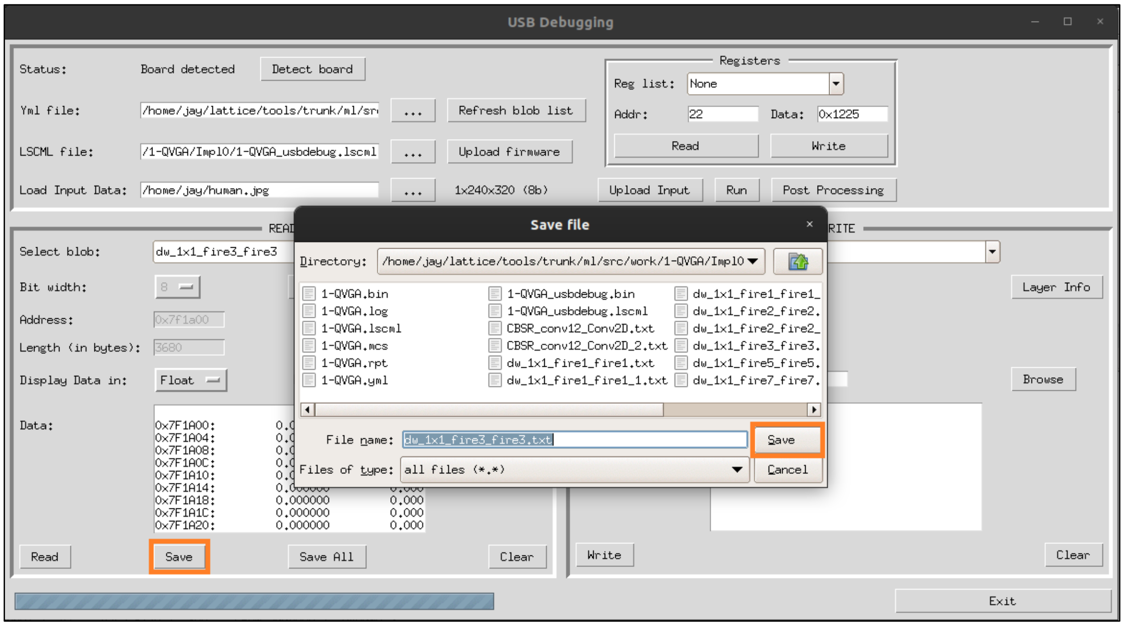


Figure 7.17. Save USB Data

Expected values for a given USB debug input are stored in the expected folder of the sensAI project directory.

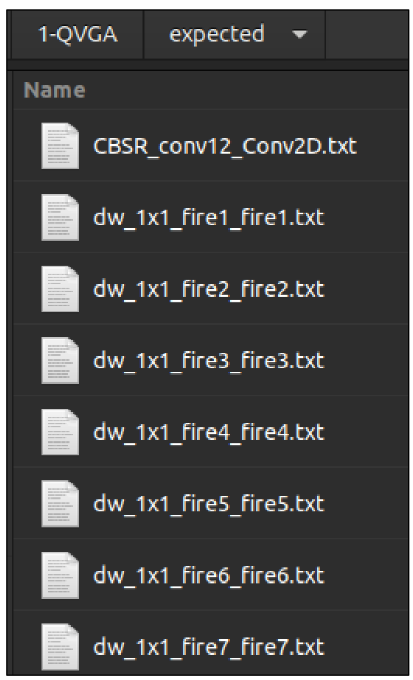


Figure 7.18. Expected Values for Corresponding Blob

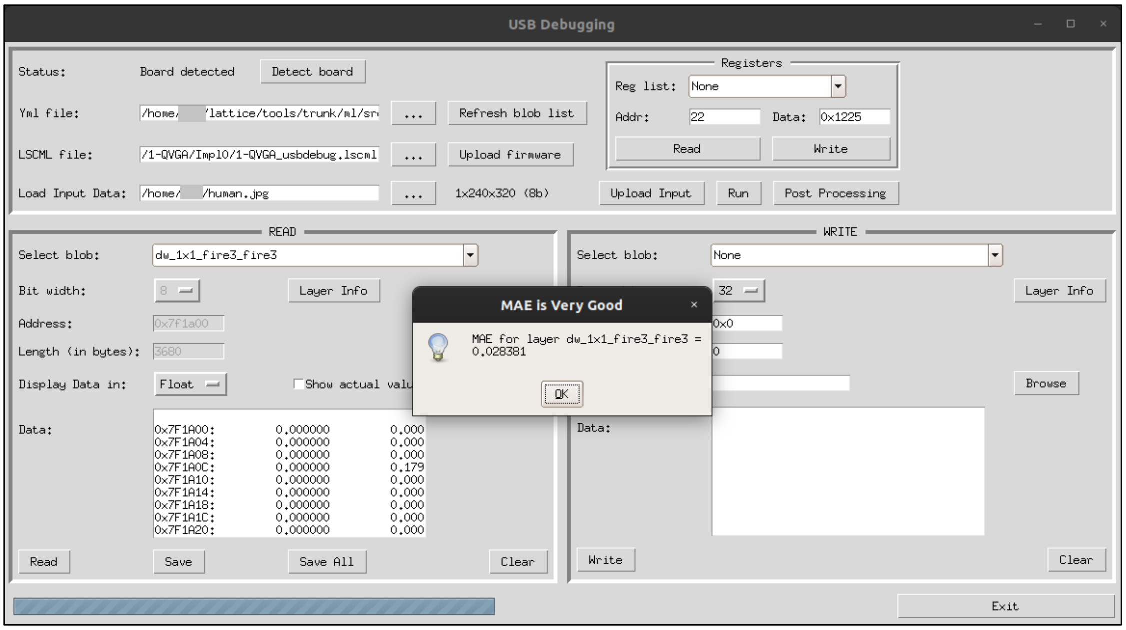


Figure 7.19. Show Expected vs HW MAE

8. RISC-V Register Interface Generator

The RISC-V register interface facilitates creation of a register file which allows communication between RISC-V and machine learning hardware. Using this interface, you can access the control and status interfaces of the ML IP.

The section provides a guide on using the RISC-V register interface generator.

8.1. Launch RISC-V System Generator Environment

To open the system generator interface, select **Tools > RISC-V System Generator**.

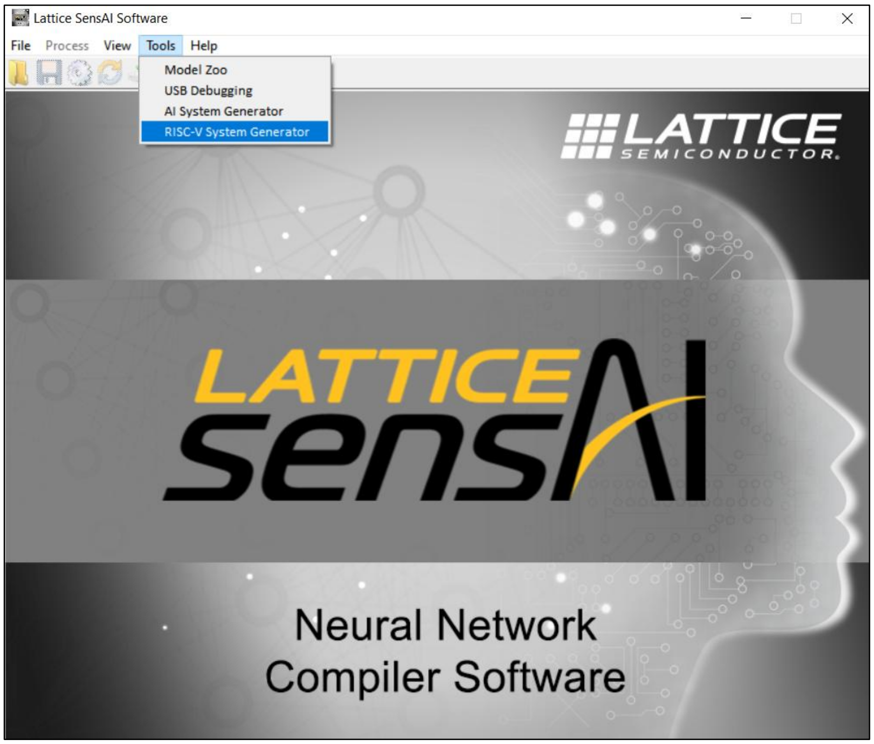


Figure 8.1. Opening the RISC-V System Generator

8.2. Generate CSR Register IP Cores

One of the core functionalities of the system generator is to generate RTL, C driver, and IPK files.

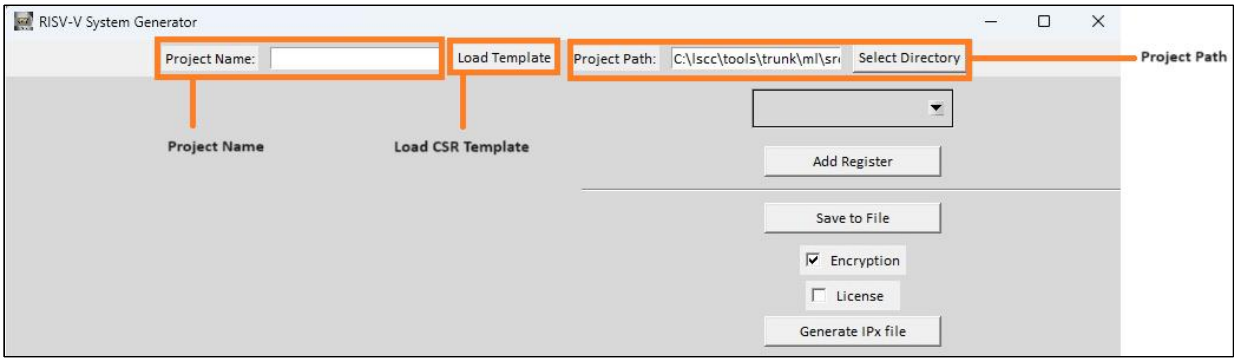


Figure 8.2. System Generator Home Window

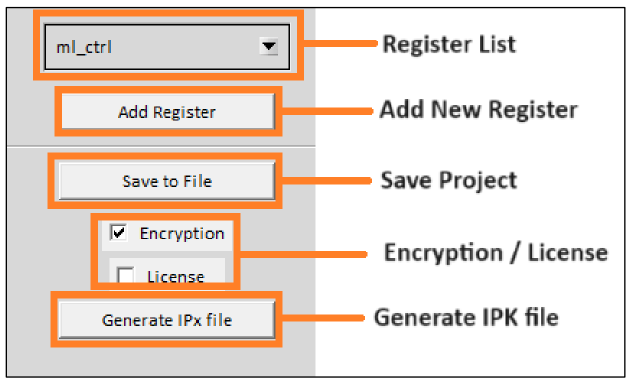


Figure 8.3. System Generator Functions

- **Register List:** List of register names available. Hover over a register name in the list to visualize the bit fields to be created in the register fields section.
- **Add New Register:** Click **Add Register**. The Add Register window appears to prompt for the register name.

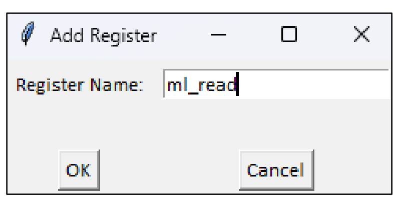


Figure 8.4. System Generator Add New Register

After entering a register name and clicking **OK**, register data and fields appear on the left. You can add more fields by clicking on the **+** button.

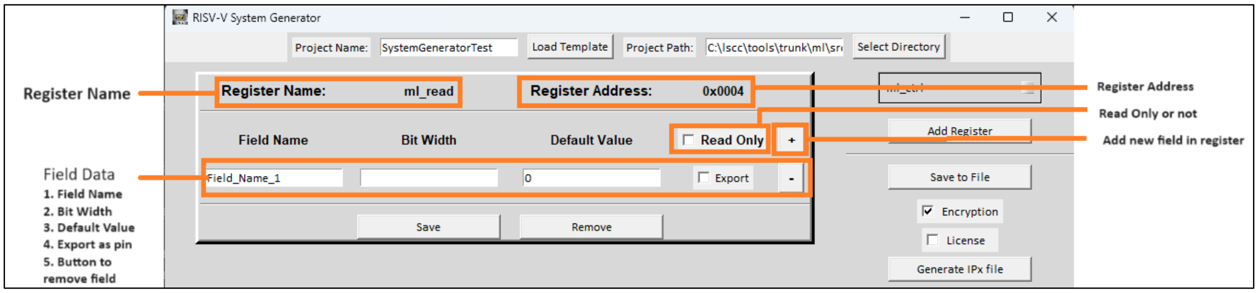


Figure 8.5. System Generator Add and Remove Register Field

Note: The total combined bits of a register should not exceed 32. Otherwise, an error message appears as shown in Figure 8.6.

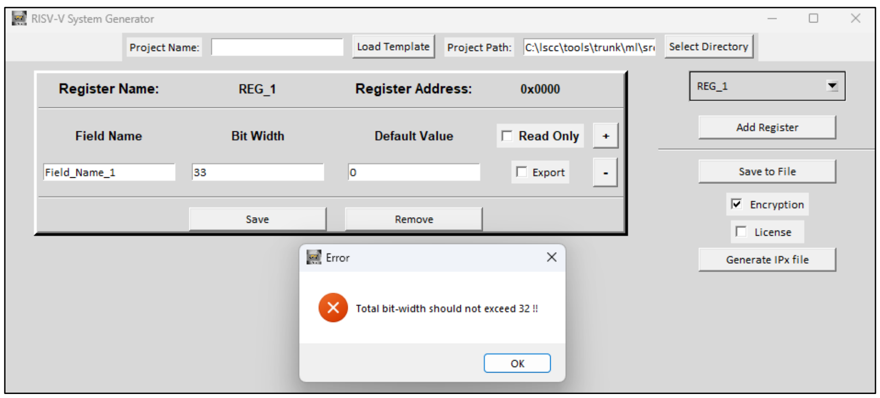


Figure 8.6. System Generator Register Bit Width Limitation

You can load CSR data from a predefined template through the **Load Template** button. The following figures shown an example template and the register data and fields after loading the template.

Address	Field	Bit index	Access	reset value	Comments	internal read	internal write
0x0000	RISCV_INPUTS_1						
	riscv_general_sel	[14:7]	RO	0	0-off 1=on	RI	
	riscv_img_in_mode	[6:6]	RO	0	0-off 1=on	RI	
	riscv_cam_done	[5:5]	RO	0	0-off 1=on	RI	
	riscv_cap_done	[4:4]	RO	0	0-off 1=on	RI	
	riscv_sc_done	[3:3]	RO	0	0-off 1=on	RI	
	riscv_ml_done	[2:2]	RO	0	0-off 1=on	RI	
	riscv_comp_start	[1:1]	RO	0	0-off 1=on	RI	
	riscv_cap_stable_img	[0:0]	RO	0	0-off 1=on	RI	
0x0004	RISCV_OUTPUTS_1						
	riscv_comp_done	[4:4]	RW	0	0-off 1=on	R	
	riscv_ml_start	[3:3]	RW	0	0-off 1=on	R	
	riscv_sc_start	[2:2]	RW	0	0-off 1=on	R	
	riscv_cap_start	[1:1]	RW	0	0-off 1=on	R	
	riscv_cam_start	[0:0]	RW	0	0-off 1=on	R	
0x0008	RISCV_OUTPUTS_2						
	riscv_ml_base_addr	[31:0]	RW	0	0-off 1=on	R	
0x000C	RISCV_OUTPUTS_3						
	riscv_sc_ibox_r	[23:12]	RW	0	0-off 1=on	R	
	riscv_sc_ibox_l	[11:0]	RW	0	0-off 1=on	R	
0x0010	RISCV_OUTPUTS_4						
	riscv_sc_ibox_u	[23:12]	RW	0	0-off 1=on	R	
	riscv_sc_ibox_b	[11:0]	RW	0	0-off 1=on	R	

Figure 8.7. System Generator Example CSR Template

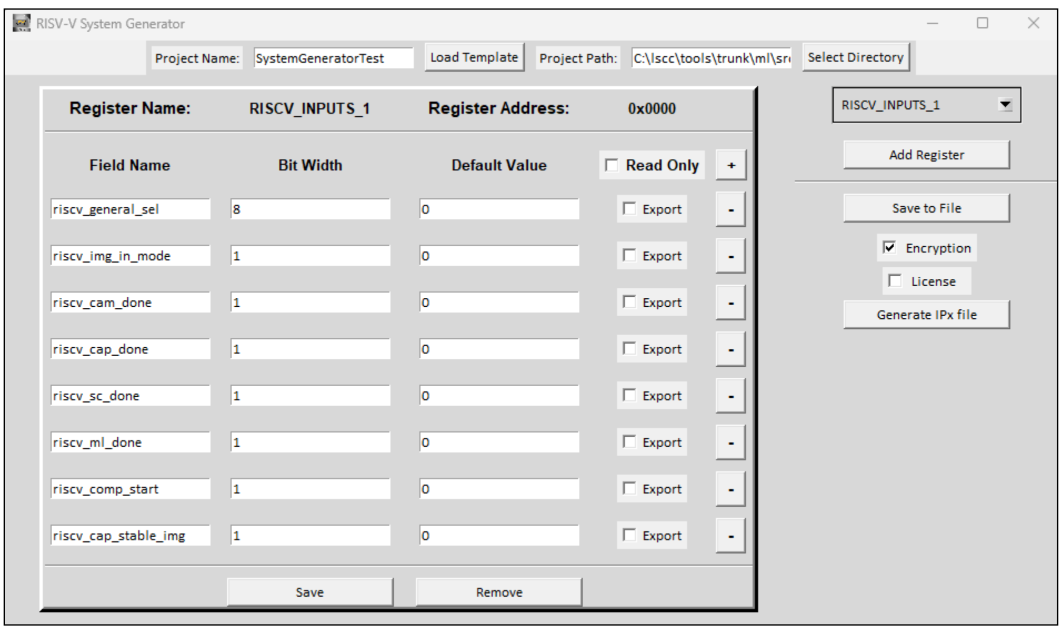


Figure 8.8. CSR Register Example

- **Save Project**: After all registers are defined, save the project file.

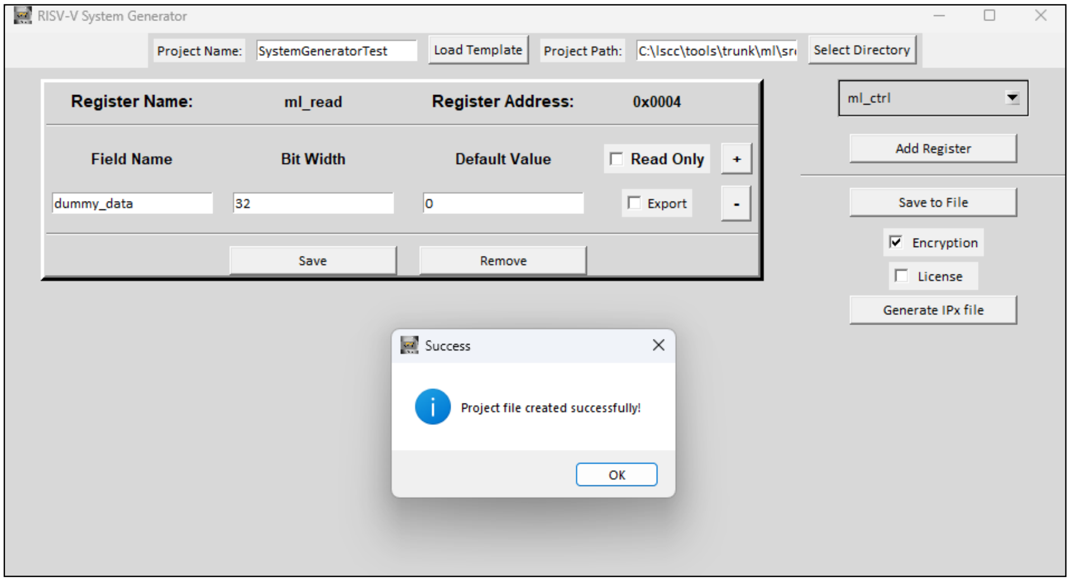


Figure 8.9. System Generator Save Project

- **Generate IPK File**: To generate the IPK file, click **Generate IPx file.** The resultant IPK file is saved under the project directory.
  - **Encryption** – Select to encrypt based on a known key from Propel so that Propel can decrypt automatically.
  - **License** – Select to include license file in the generated IPK file.

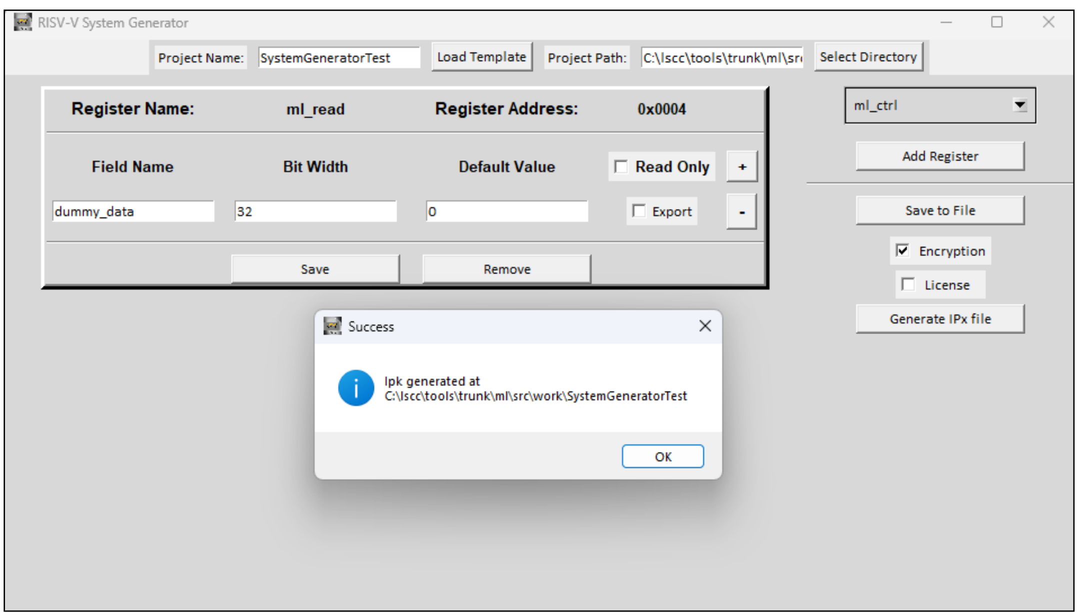


Figure 8.10. System Generator Generate IPK File

Limitations:

- The system generator can be enabled for Advanced CNN IP on the Windows platform only.
- To use the RISC-V CSR Register IP generation feature, the system (host machine where the SensAI SDK is running) must meet the requirements mentioned in Radiant Installation (Lattice Radiant Software 2025.1 for Linux Ubuntu or Lattice Radiant Software 2025.1 for Windows) and Propel Installation.
- If a space exists in a username, RISC-V CSR Register IP generation will fail. If this occurs, move your build to a location where the path does not contain the username, for example *c:/lscc/sensai*.

Appendix A. Supported Keras Layers

In general, the supported Keras layers need to be similar to the supported TensorFlow operations in compute topology, as described in Appendix C. Supported TensorFlow Operations, and have the same hardware constraints and parameter requirements.

This appendix currently only lists supported Keras layers without additional commentary. See the Keras demo designs in the sensAI network directory and refer to the chapters on TensorFlow and Caffe for more information on how to utilize these layers in your own designs.

The layers supported for AutoKeras are same as Keras layers.

- Conv2D
    - To perform 8-bit weight quantization in Keras, refer to the Fixed Point Quantization Training in Keras section for details on implementation.
- BatchNormalization
- Dense
- MaxPooling2D
- AveragePooling2D
- DepthwiseConv2D
- Input
- Lambda (only for 8-bit activation quantization)
    - We use the Lambda function for 8-bit quantization of activation in Keras. Refer to the Fixed Point Quantization Training in Keras section for details on implementation. Please note that the Lambda function is dependent on the version of Python, and you might face issues regarding Marshal Data if the training and inferencing environments are different. Hence, it is advised that if the trained Keras model by the user has a Lambda function for activation quantization, convert the Keras model to Tensorflow in the same training environment. For this conversion, as a reference, you can refer to the ***ReferenceDesign/Training/keras-to-tf-converter*** folder of this Reference Design.
- LeakyReLU
- ReLU
- Split
- Concatenate
- GEMM(MatMul + Mul)
- Add (for elementwise addition)
- UpSampling2D/Resizing

  Support nearest and bilinear for argument interpolation. We only support up scaling by 2 in spatial dimensions.
- Silu (Also known as Swish)
    - Input quantization range varies from –8 to 8. If the layer is intermediate, then output will be quantized with 8 bits. If the layer is final i.e. last layer, then no need to quantize output since QuantizeOuput layer will be added at the last with 16 bits quantization. Silu is LUT based function
- Sigmoid/Softmax

Input quantization range varies from –4 to 4. If the layers are intermediate, the output will be quantized to 8 bits. If the layer is the final layer (that is, the last layer), there is no need to quantize the output because a Quantize Output layer will be added at the end with 16-bis quantization. Sigmoid/Softmax is a LUT-based function. Higher input/output precision requires higher hardware resources.

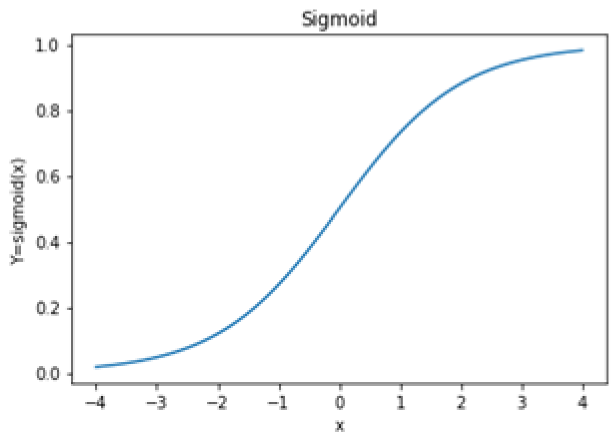


Figure B.1. Sigmoid Function

Other than these layers, we use native TensorFlow operators in Keras to perform some of the operations in the Lattice Neural Network Compiler for post-processing only. Following is a list of those operations and their purposes:

- Tf.math.multiply : For scalar multiplication or eltwise multiplication with one constant tensor as a second operator
- Tf.math.subtract : For scalar subtraction or eltwise subtraction with one constant tensor as a second operator
- Tf.math.add: For scalar addition or eltwise addition with one constant tensor as a second operator
- Tf.math.reciprocal_no_nan : For reciprocal operation on the input tensor.
- Tf.math.power : For the power operation, which currently supports the powers of two.
- Tf.strided_slice : This operation is used either alone for the strided_slice operation or with the Concat layer to implement the focus layer. While implementing strided slice, except for begin indices, no other indices can have 0. See the example below for how to use strided slices and implement the focus layer.

```
def example_focus_layer(inputs, input_shape=None):
    if input_shape:
        out = []
        H,W,C = input_shape
        out.append(tf.strided_slice(inputs, [0, 1, 1, 0], [1, 160, 256, 3], [1, 2, 2, 1]))
        out.append(tf.strided_slice(inputs, [0, 0, 1, 0], [1, 160, 256, 3], [1, 2, 2, 1]))
        out.append(tf.strided_slice(inputs, [0, 1, 0, 0], [1, 160, 256, 3], [1, 2, 2, 1]))
        out.append(tf.strided_slice(inputs, [0, 0, 0, 0], [1, 160, 256, 3], [1, 2, 2, 1]))
        input_focus = Concatenate(axis=3)(out)
```

Figure B.2. Strided Slice Example

Appendix B. Supported Layer Configuration

This appendix provides information on parameter configuration for each layer based on device type or mode.

Table B.1. Supported Layer Configuration

Layer Name	Parameter	Device Type, Mode, and IP						
		Optimized CNN	Compact CNN	Extended CNN	Advanced CNN	iCE40 UltraPlus	ECP5 – Dual	ECP5 – Mobilenet
Convolution	Kernel size	3 × 3	3 × 3	3 × 3	3 × 3	Up to 3 × 3	Up to 9 × 9	Up to 9 × 9
	Pad	0 or 1	0 or 1	0 or 1	1	1	1	1
	Stride	1 or 2	1	1	1 or 2	1	1	1
	Kernel size (5 × 5)	Not supported	Not supported	Not supported	5 × 5	Not supported	Not supported	Not supported
	Pad				2			
	Stride				1			
	Kernel size (7 x 7)	Not supported	Not supported	Not Supported	7 x 7	Not supported	Not supported	Not supported
	Pad				3			
	Stride				1			
Depthwise Convolution	Kernel size	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	N/A	3 × 3
	Pad	0 or 1	0 or 1	0 or 1	1	1		0 or 1
	Stride	1 or 2	1	1	1 or 2	1		1 or 2
1×1 Convolution	Kernel size	1 × 1	1 × 1	1×1	1 × 1	1 × 1	N/A	1 × 1
	Pad	0	0	0	0	0		0
	Stride	1	1	1	1	1		1
Binary Convolution	Kernel size	Not supported	Not supported	Not supported	Not supported	3 × 3	3 × 3	Not supported
	Pad					1	1	
	Stride					1	1	
Batch Normalization	scale, bias, mean, variance	Supported	Supported	Supported	Supported	Supported	Supported	Supported
Max Pooling	Kernel	2 × 2	2 × 2	2 × 2	2 × 2	2 × 2	Must be symmetric	Must be symmetric
	Stride	2	2	2	2	2	1	1
	Pad	0	0	0	0	0	0	0
Max Pooling K × K	Kernel	K × K	Not supported	Not supported	K × K	Not supported	Not supported	Not supported
	Stride	1			1			
	Pad	K//2			K//2			
Global Average Pooling	Kernel	Must be symmetric	Not supported	Not supported	Must be symmetric	Not supported	Must be symmetric	Not supported
	Stride	1			1		1	
	Pad	0			0		0	
Argmax Pooling	Kernel	Not Supported	Not Supported	2 × 2	2 × 2	Not Supported	Not Supported	Not Supported
	Stride			2	2			
	Pad			0	0			

Layer Name	Parameter	Device Type, Mode, and IP						
		Optimized CNN	Compact CNN	Extended CNN	Advanced CNN	iCE40 UltraPlus	ECP5 – Dual	ECP5 – Mobilenet
Leaky ReLU Negative slope	Training Param Alpha	0.0625 (1/16)	0.0625 (1/16)	0.0625 (1/16)	0.0625 (1/16)	0.0625 (1/16)	0.0625 (1/16)	0.0625 (1/16)
Sigmoid	Input bits	1 to 16	Not Supported	Not Supported	16	Not Supported	Not Supported	Not Supported
	Output bits	8 or 16			8			
	MSB clip enable	0 or 1			0 or 1			
Softmax / Silu	Input bits	Not supported	Not supported	Not supported	16	Not support ed	Not Supported	Not Supported
	Output bits				8			
	MSB clip enable				0 or 1			
Fully Connected / Dense	Number of inputs	Any (Must be last layer)	Any (Must be last layer)	Any (Must be last layer)	Any (Must be last layer)	≤ 1024	Any	Any
Elementwise Addition	N/A	Supported	Supported	Supported	Supported	Not Supported	Supported	Supported
Elementwise Subtraction	N/A	Not Supported	Not Supported	Not Supported	Supported	Not Supported	Not Supported	Not Supported
Multiplication	N/A	Not Supported	Not Supported	Not Supported	Supported	Not Supported	Not Supported	Not Supported
Focus	N/A	Supported	Not Supported	Not Supported	Supported	Not Supported	Not Supported	Not Supported
Dilated Convolution	Dilation Parameter	Not supported	Not supported	2 or 4	Not supported	Not supported	Not supported	Not supported
Resize/Upsampl e	Bilinear	Supported	Not supported	Supported	Supported	Not supported	Not supported	Not supported
	Nearest	Supported	Not supported	Not supported	Supported	Not supported	Not supported	Not supported
Unpooling	Kernel	Not supported	Not supported	2 × 2	Not supported	Not supported	Not supported	Not supported
	Stride			2				
	Pad			0				
Split	Channels to split	Supported	Not supported	Not Supported	Supported	Not supported	Not supported	Not supported
Concat	Channels to concat	Supported	Not supported	Not Supported	Supported	Not Supported	Not Supported	Not Supported
GEMM	N/A	Not Supported	Not Supported	Not supported	Supported	Not Supported	Not Supported	Not Supported

Appendix C. Supported TensorFlow Operations

This appendix is intended to provide information for TensorFlow operations currently supported. SensAI supports TensorFlow versions 2.9, 2.5, 2.3, 2.0, and 1.14, which are the versions used to test Network Compiler.

Batch Normalization

Currently, Rsqrt is the operation tag used to locate and analyze the batch normalization subgraph (a group of operations), based on the tf.nn.batch_normalization implementation. Therefore, the software does not support the model where Rsqrt is used in the graph but not for batch normalization. If you do not use tf.nn.batch_normalization to create a batch normalization subgraph, the batch normal subgraph should be in the same computation order and structure, as shown in the following Figure C.1. If variance epsilon (*y* in Figure C.1) of batch normalization is not provided, the default value 1e-3 should be used. If the offset (*beta* in Figure C.1) is not provided, the default value of *0.0* should be used.

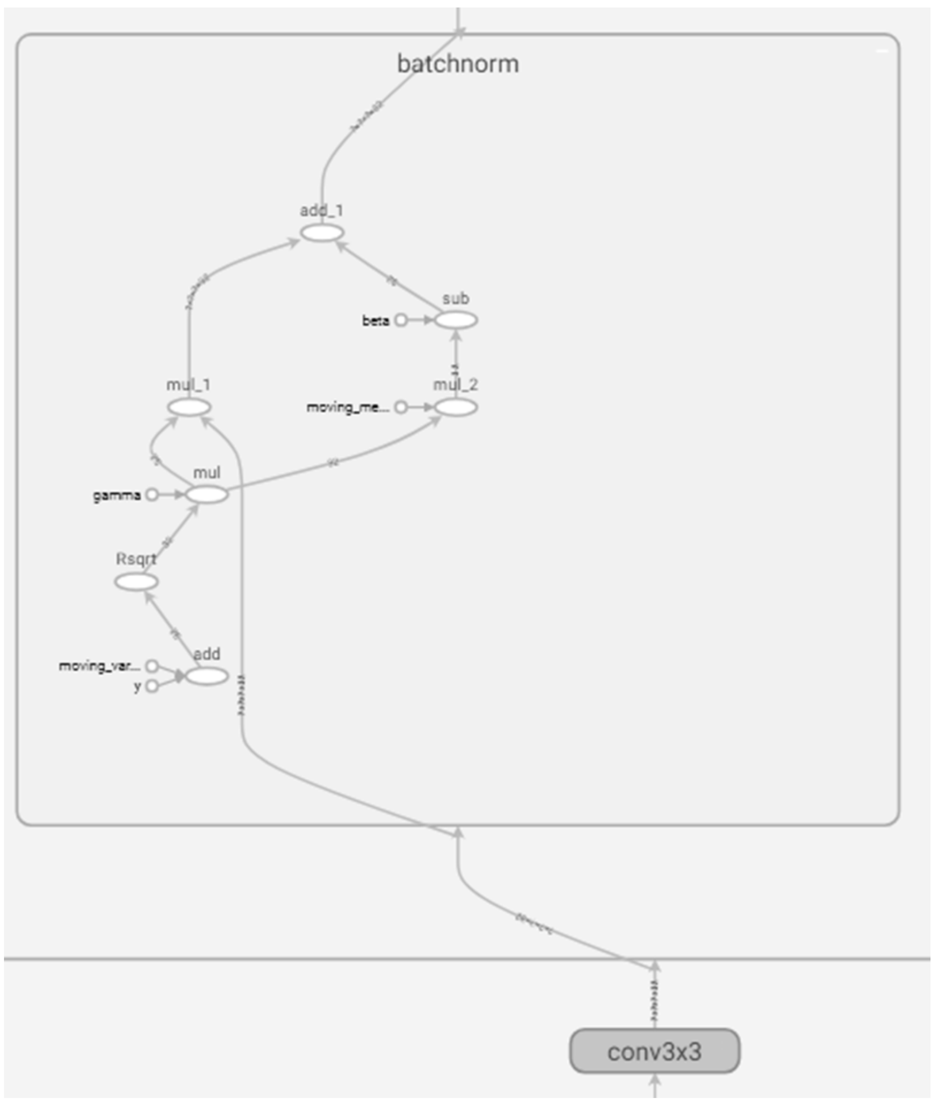


Figure C.1. Batch Normalization

An optimized implementation such as fused batchnorm is also supported.

Conv2D

The software only supports regular Conv2D. The Conv2D node is required to be the bias node's (BiasAdd) direct input in order to apply the bias to the Conv2D layer. Other convolution operations, such as stride > 1, are not generally supported.

DepthwiseConv2dNative, dilated convolution, and quantized convolution are supported in certain topology contexts. For quantized convolution, refer to the Fixed Point Quantization Training in TensorFlow section. If you are creating a Conv2D layer with stride 2, it is recommended not to use an explicit padding layer just before Conv2D. Instead, use the padding option within the Conv2D layer such that the padding is asymmetric.

Channel Padding

Channel padding refers to the operation where the input tensor is padded with zeros on the channel dimension to increase the number of channels. This is performed by using the tf.Pad operation.

Concat

This is performed by using the tf.concat operation.

Elementwise Add

Elementwise Add is only supported when being used in residual net, with two tensor objects as the only input where the coefficients for each are 1. In general, low-level elementwise operations such as, mul, div, sub, max, etc. are not supported.

Matmul

Matmul is only supported in regular, fully connected, or dense layers. Advanced transpose, and adjoint mode are supported. Sparse is not supported. We don't support complex type and unofficial operations, TF contributions, customized open source, such as tf.contrib.layers.fully_connected, implementations are not supported.

Placeholder

Support is limited to inputs with a standard 4 or 3 dimension shape for images and 2 or 1 dimension for audio. Only one placeholder can exist in the optimized frozen inference graph. Preprocess operations on input are not supported. The expected input is a single image, gray or color, after preprocessing. Group image and video formats are not supported.

Pooling

The software currently supports three types of Pooling:

- Maxpool: tf.nn.max_pool
- Global Average Pooling: tf.reduce_mean
- MaxPoolWithArgMax : tf.nn.maxpool_with_argmax

ResizeBilinear

We use the ResizeBilinear operation to perform upsampling, replacing the deconvolution operation in encoder-decoder like network topologies by using tf.image.resize_bilinear. And this implementation uses half_pixel_centers as true. So far, the operation is supported only during segmentation.

ResizeNearestNeighbour

We use the ResizeNearestNeighbor operation to perform upsampling, replacing the deconvolution operation in encoder-decoder like network topologies by using tf.image.resize_nearest_neighbor And this implementation uses asymmetric as true. So far, the operation is supported only during segmentation.

Unpool

Unpooling is the opposite operation of pooling. This operation uses one of the outputs of MaxPoolWithArgMax, max indices, and performs unpooling with the help of multiple operations. The implementation example can be seen below.

```
def unpool(updates, mask, k_size=[1,2,2,1], output_shape=None, scope=""):
    with tf.variable_scope(scope):
        mask = tf.cast(mask,tf.int32)
        input_shape = tf.shape(updates, out_type=tf.int32)
        # Calculation enw shape
        if output_shape is None:
            output_shape = (input_shape[0], input_shape[1]*k_size[1],
input_shape[2]*k_size[2], input_shape[3])

        # Calculation indices for batch, height, width and feature maps
        one_like_mask = tf.ones_like(mask, dtype=tf.int32)
        batch_shape = tf.concat([[input_shape[0]],[1],[1],[1]],0)
        batch_range = tf.reshape(tf.range(output_shape[0],dtype=tf.int32),
shape=batch_shape)
        b = one_like_mask * batch_range
        y = mask//(output_shape[2]*output_shape[3])
        x = (mask//output_shape[3])%output_shape[2]
        feature_range = tf.range(output_shape[3],dtype=tf.int32)
        f = one_like_mask * feature_range

        # Transpose indics & reshape update values to one dimension
        updates_size = tf.size(updates)
        indices = tf.transpose(tf.reshape(tf.stack([b,y,x,f]),[4,updates_size]))
        values = tf.reshape(updates, [updates_size])
        ret = tf.scatter_nd(indices,values,output_shape)
        return ret
```

Figure C.2. Unpool Implementation

ReLU

The software currently only supports normal ReLU, which is implemented by tf.nn.relu (slope = 1 in the positive region and slope = 0 in the negative region).

For leaky ReLU (non-zero alpha slope in the negative region), sensAI supports tf.nn.leaky_relu and customized implementations based on tf.nn.relu. For example, tf.nn.relu(x) - alpha * tf.nn.relu(-x). The negative activation slope for leaky_ReLU in a model must be fixed to 1/16, corresponding to alpha = 0.0625. Leaky ReLU is only supported on ECP5 devices.

Sigmoid

This is performed by using the tf.nn.sigmoid operation.

Softmax

This is performed by using the tf.nn.softmax operation.

Silu

This is performed by using mul and sigmoid operation. Silu(input) = input * sigmoid(input)

Appendix D. USB Debugging Register Map

The following are the registers that can be read or write using the sensAI USB debugging interface.

Table E.1. USB Debugging Register Map

Address	Register Name	RW mode	Default value	Description
0x0000	dev_type_ver	RO	0x00010001	Indicates device type and version.
0x0010	gp_ctl00	RW	0x00000000	Bit[4]: continuous run. Bit[0]: single run.
0x0011	gp_ctl01	RW	0x00000000	Bit[8]: vid_reset Bit[0]: automatic gain control enable.
0x0012	gp_ctl02	RW	0x00000000	—
0x0013	gp_ctl03	RW	0x00000000	—
0x0014	gp_ctl04	RW	0x00000000	—
0x0020	gp_status00	RO	0x00000000	Bit[8]: single run request. Bit[7:0] ml_status
0x0021	gp_status01	RO	0x00000000	Number of cycles.
0x0022	gp_status02	RO	0x00000000	Number of commands.
0x0023	gp_status03	RO	0x00000000	Number of cycle for DMA access.
0x0024	gp_status04	RO	0x00000000	Number of DMA commands.
0x0025	gp_status05	RO	0x00000000	Number of loss time due to fifo underrun.
0x0026	gp_status06	RO	0x00000000	Number of cycles for convolution and pooling.
0x0027	gp_status07	RO	0x00000000	Number of cycles for full connecting.
0x0028	gp_status08	RO	0x00000000	GPO value
0x0029	gp_status09	RO	0x00000000	cycle for LDMA access (Only for CPNX advanced IP and Avant device)
0x002a	gp_status0a	RO	0x00000000	cycle for Advanced Engine ALU operation (Only for CPNX advanced IP and Avant device)
0x002b	gp_status0b	RO	0x00000000	cycle for scale operation (Only for CPNX advanced IP and Avant device)
0x002c	gp_status0c	RO	0x00000000	cycles of waiting (Only for CPNX advanced IP and Avant device)
0x0030	ba_code	RW	0x00000000	Base address for firmware.
0x0031	ba_input	RW	0x0f000000	Base address for input data (iCE40 UltraPlus device only).
0x0032	ba_output	RW	0x0f100000	Base address for output data (iCE40 UltraPlus device only).
0x0100	reg_waddr	RW	0x00000000	AXI write address.
0x0101	reg_wconf	RW	0x00000000	AXI write configure.
0x0110	reg_raddr	RW	0x00000000	AXI read address.
0x0111	reg_rconf	RW	0x00000000	AXI read configure.
0x0200	sw_i2c	RW	0x00000003	Software controlled I2C interface.
0x0300	hw_i2c_conf	RW	0x00000000	Hardware I2C master configure.
0x0301	hw_i2c_status	RO	0x00000000	Hardware I2C master status.
0x0302	hw_i2c_pack	RW	0x00000000	Bit[31:16]: I2C address. Bit[15:0]: I2C write data.
0x0303	hw_i2c_rdata	RO	0x00000000	Hardware I2C master data configure.

Appendix E. Supported ONNX Layers

The ONNX layers need to be similar to the supported TensorFlow operations in the compute topology as described in Appendix B. Supported Layer Configuration. Supported ONNX operations have the same hardware constraints and parameter requirements. As support is experimental, some layers or attributes might not be supported.

Appendix F. Network Topology and Device Table

The following table lists all known supported network topologies and the devices that support them. For more details about layer restrictions, device restrictions, and required or suggested network implementation options, consult the Getting Started section and the Advanced Topics section.

Boxes in green indicate a network/device combination available as part of Lattice's Model Zoo, except for GoogleNet and Squeezedet.

Table F.1. Network Topology and Device

Network	ECP5	iCE40 UltraPlus	CrossLink-NX and CertusPro-NX
MobilenetV1	Supported - Mobilenet Mode only	Supported	Optimized and Extended IP
MobilenetV2	Supported - Mobilenet Mode only	Unsupported	Optimized and Extended IP
ResNet	Supported	Unsupported	Optimized and Extended IP
SSD	Supported – Dual engine mode	Unsupported	Optimized and Extended IP
tinyVGG	Supported	Supported	Supported
VGG	Supported	Supported	Optimized and Extended IP
YOLOv1	Supported	Unsupported	Unsupported
TinySSD	Supported	Unsupported	Unsupported
MobileNetv2-SSD	Unsupported	Unsupported	Optimized and Extended IP
GoogleNet	Unsupported	Unsupported	Optimized and Extended IP
SqueezeDet	Unsupported	Unsupported	Optimized and Extended IP
Enet	Unsupported	Unsupported	Extended IP
Yolov5	Unsupported	Unsupported	Optimized and Advanced IP
YoloV8	Unsupported	Unsupported	Optimized IP
Yolov11	Unsupported	Unsupported	Only on Advanced mode IP
ResNet V1	Unsupported	Unsupported	Optimized and Advanced IP
18	Unsupported	Unsupported	Optimized and Advanced IP

Note: Some modifications are required for models to meet device or layer restrictions and to enable support it in the NNC compiler.

Appendix G. Common CNN Blocks Used in Lattice NNC

This section shows how common modules and blocks used in CNN architectures are customized for Lattice NNC. For detailed information about each module parameter refer to the restriction sections of the particular device any model is run on.

Generic Blocks

The following are some of the generic modules used in our compiler.

- Relu refers to Relu2 in all the blocks in this and the next sections.
- Bias in convolution is supported only for ECP5.
- In the majority of cases, the convolution block will be followed by BatchNorm (with scale), QuantRelu (device-specific), and Relu. This structure, from here on, is referred to as CBSR.
- Generally, instead of using CBSR with Stride 2 (SAME padding), we use CBSR with Stride 1 (SAME padding), followed by MaxPool2D with Kernel 2 and Stride 2.
- For all the next sections in x.x. , in all the diagrams, ***Q*** will be used for quantized and ***N*** will refer to Non-quantized.

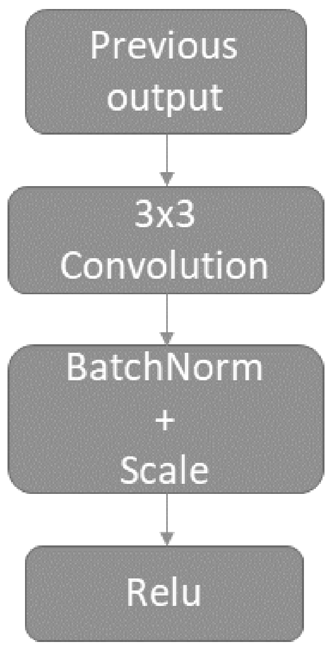


Figure H.1. Non-Quantized 3x3 CBSR or 3x3 Depthwise CBSR

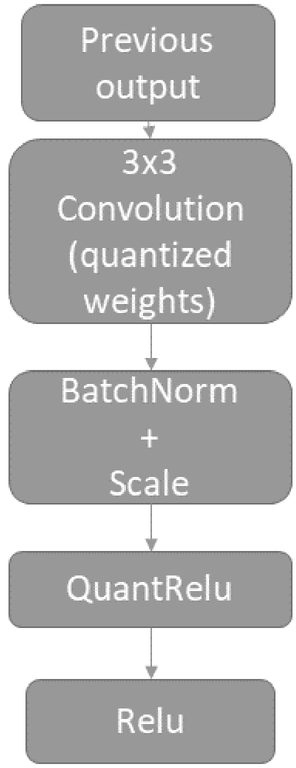


Figure H.2. Quantized 3x3 CBSR or 3x3 Depthwise CBSR

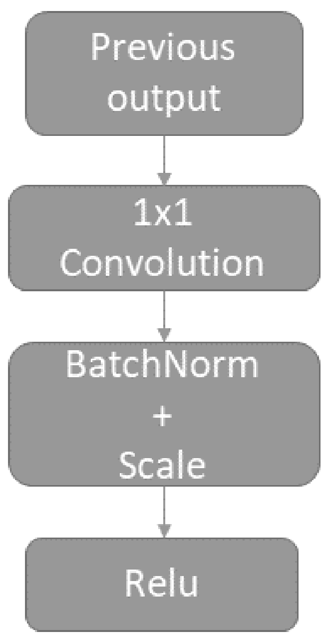


Figure H.3. Non-Quantized 1x1 CBSR

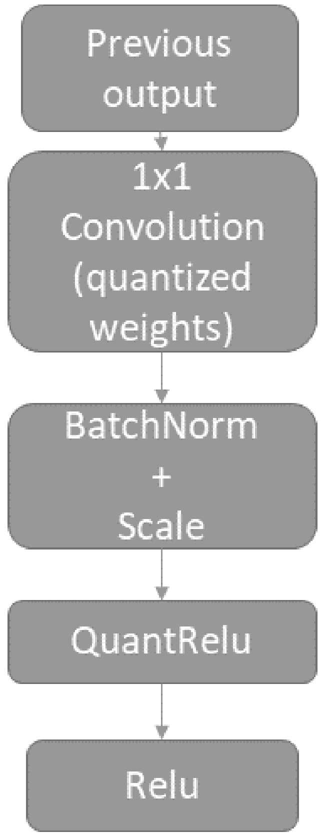


Figure H.4. Quantized 1x1 CBSR

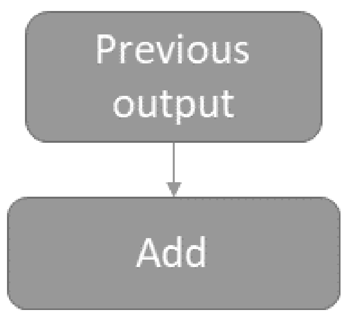


Figure H.5. Non-Quantized Add Block

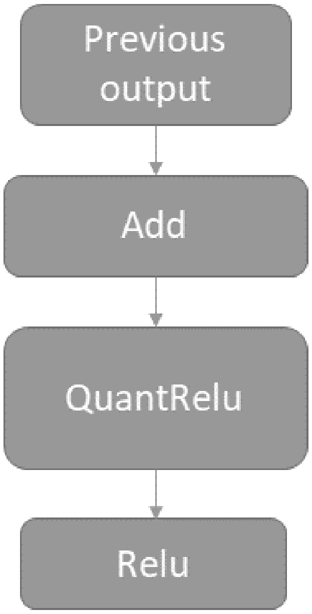


Figure H.6. Quantized Add Block

VGG

For some devices (for classification), only a single dense layer is supported at the end instead of multiple dense layers.

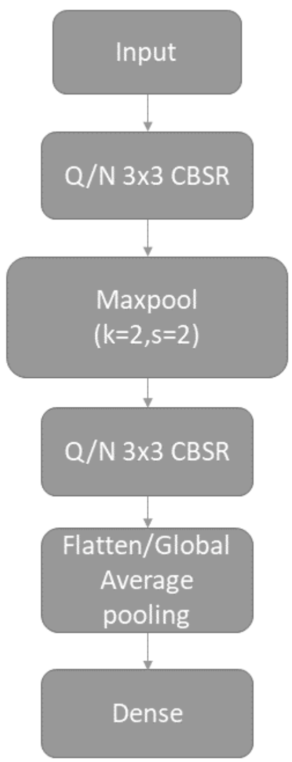


Figure H.7. VGG toy model

MobileNetV1

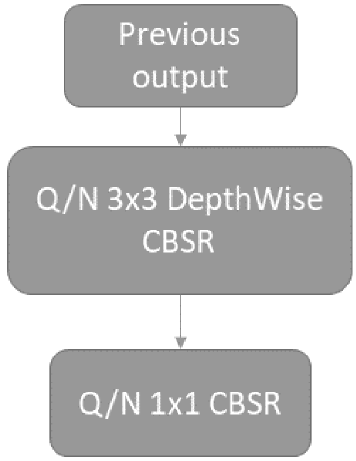


Figure H.8. MobileNetV1 Block

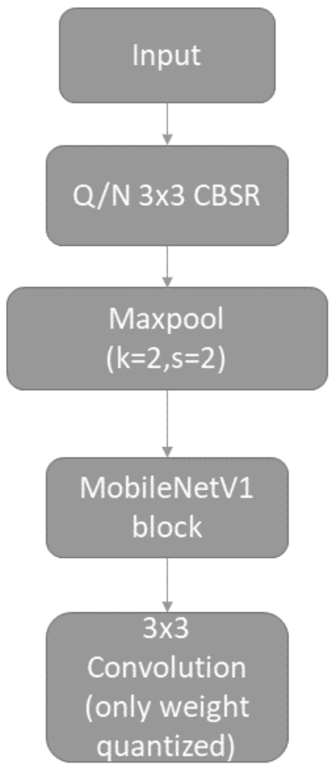


Figure H.9. MobileNetV1 Toy Model

MobileNetV2

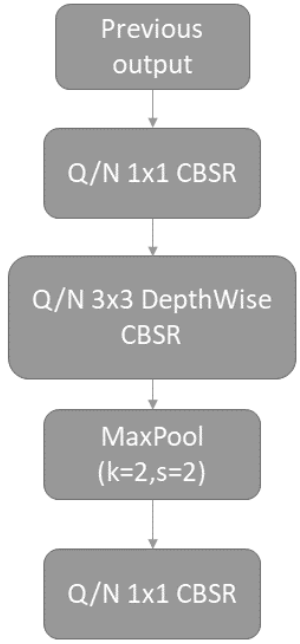


Figure H.10. MobileNetV2 Block 1

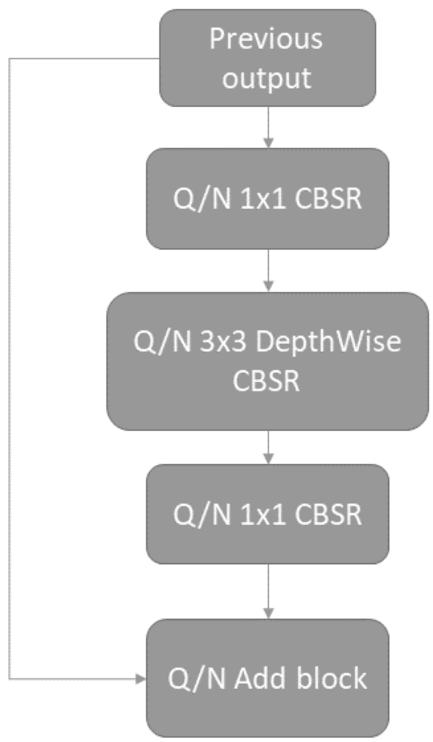


Figure H.11. MobileNetV2 Block 2

ResNet

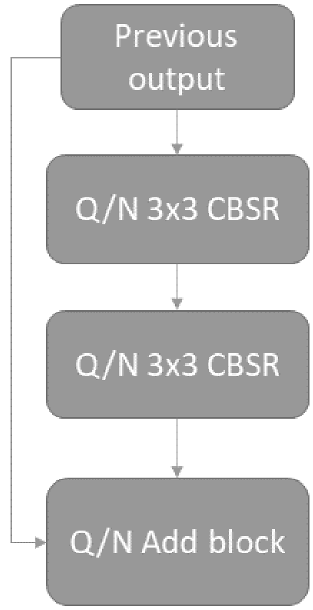


Figure H.12. ResNet Toy Model

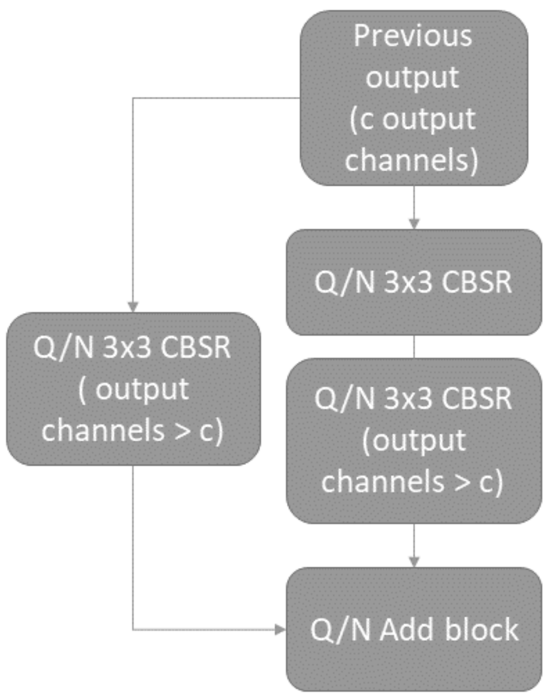


Figure H.13. ResNet Block 2 Variation 1

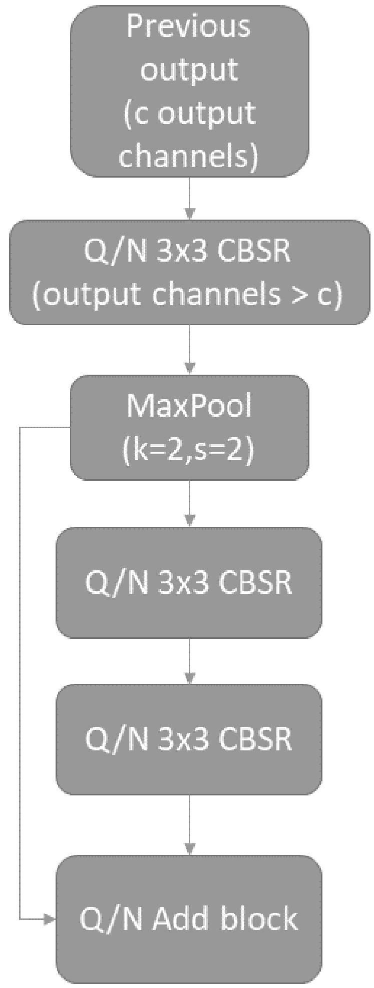


Figure H.14. ResNet Block 2 Variation 2

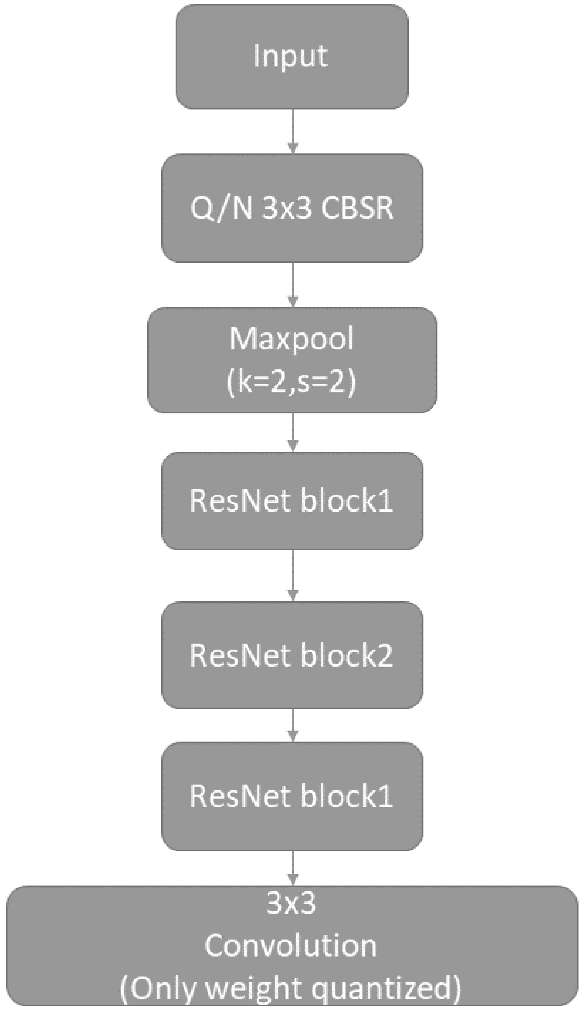


Figure H.15. ResNet Block 2 Variation 3

GoogleNet

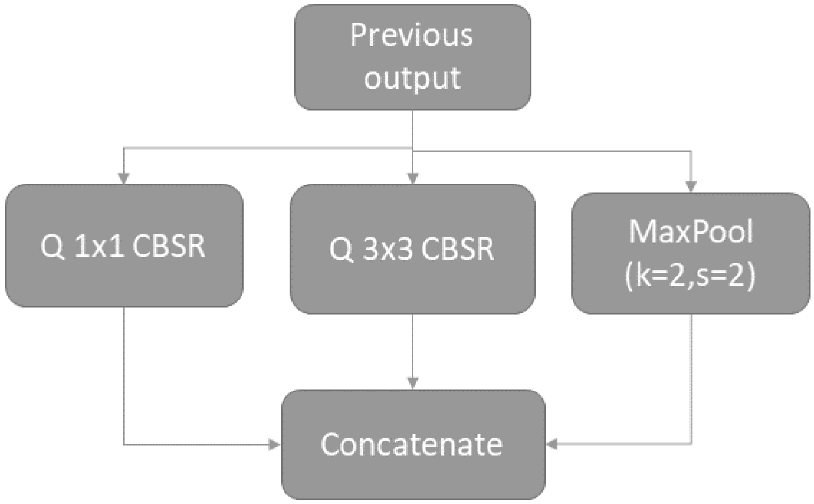


Figure H.16. GoogleNet Inception Block 1

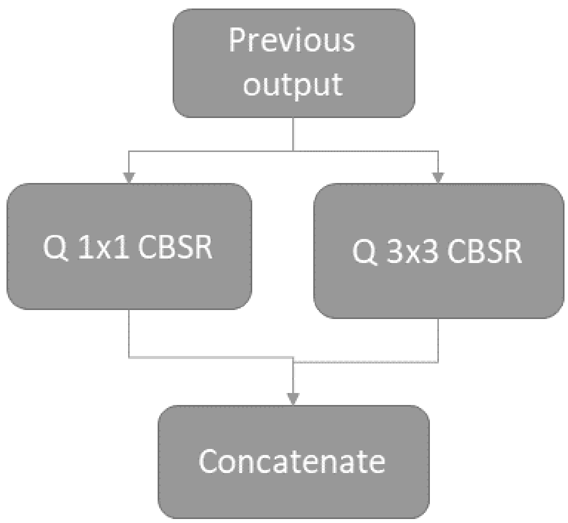


Figure H.17. GoogleNet Inception Block 2

ENET

The following figures show the four basic blocks used in ENET.

BSR in the Upsample block refers to BatchNorm + Scale + QuantRelu + Relu.

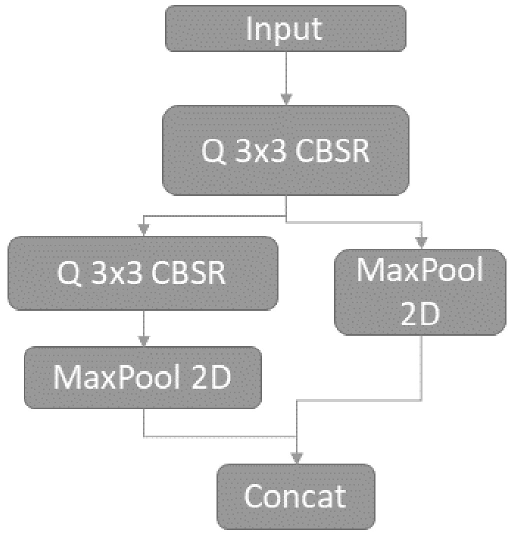


Figure H.18. Init Block

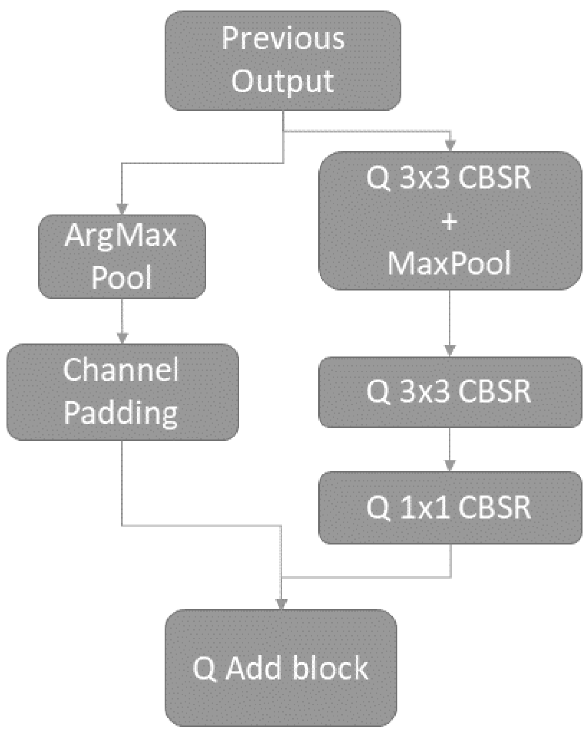


Figure H.19. DownSample Block

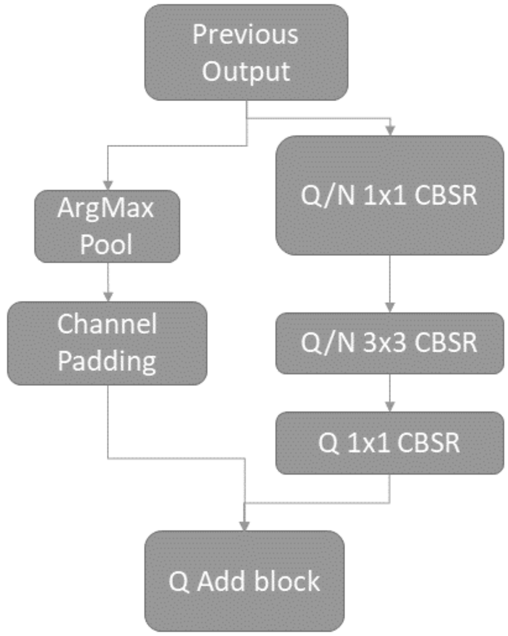


Figure H.20. Regular Block

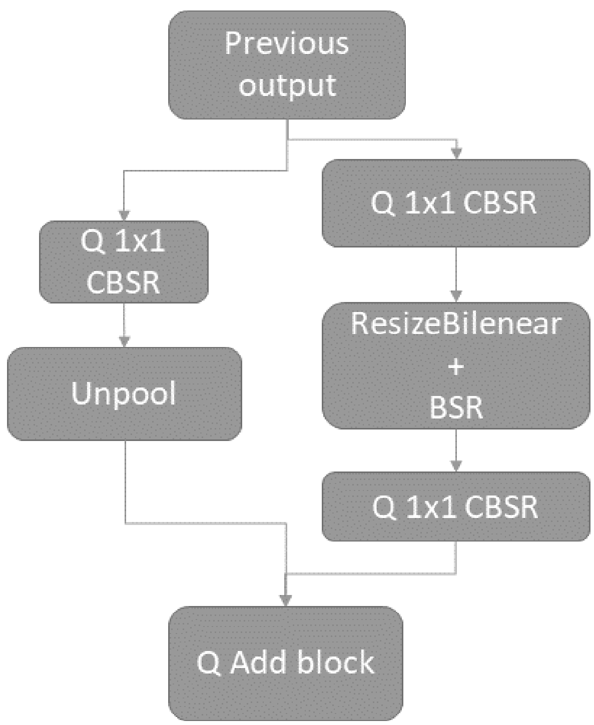


Figure H.21. Upsample Block

Table G.1. Enet Example Architecture

Type	Output Size
Input	1x160x160
Init Block	12x80x80
Downsample block	40x40x40
4xRegular Block	40x40x40
Downsample Block	80x20x20
Regular+Dilated (d=2) Blocks	80x20x20
2x(Regular+Dilated (d=4))	80x20x20
Upsample Block	40x40x40
Regular	12x80x80
ResizeBilinear + BSR	12x160x160
3x3 Convolution output	2x160x160

References

- USB3-Gigabit Ethernet Demo User Guide (FPGA-UG-02054)
- Learned Step Size Quantization paper
- Lattice sensAI Human Counting AI Demo web page
- Lattice Semiconductor GitHub
- Lattice Diamond 3.13 User Guide
- Lattice Radiant Software 2025.1 User Guide
- Lattice Diamond FPGA design software
- 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/en/Support/AnswerDatabase.

Revision History

Revision 8.0, December 2025

Section	Change Summary
Introduction	• Removed *Caffe* from sensAI SDK under the General Requirements section. • Updated the Neural Network Compiler version to *8.0* under the IP Requirements section.
Installing the Software	• Updated Machine Learning Software version to *8.0*. • Updated Figure 2.1. Installation Location Specification, Figure 2.2. Installation Component Specification, and Figure 2.3. Installation Ready to Install Dialog Box.

Getting Started	• Updated Figure 3.1. Project Settings Window, Figure 3.4. Project Implementation Options Window, Figure 3.5 Project Implementation Window 2 (Only for Advanced IP), and Figure 3.6. Project Window.
	• Removed *Caffe* from supported frameworks.
	• Updated Table 3.1. Arguments and Usage.
	  • Removed support for *Caffe*.
	  • Added option to enable/disable *VE ALU*.
	  • Removed *KMAX* pooling options.
	• Updated the UltraPlus Restriction section.
	  • Added this bullet: *Bias is not supported in convolution layer*.
	• Updated the CrossLink-NX and CertusPro-NX Optimized and Extended Mode Restrictions section.
	  • Added support for *Bias* in Convolution operation.
	  • Added support for branch structure with *Concat layer* in embedded mode.
	  • Added support for *Resize operation*.
	• Updated the CertusPro-NX and Avant Advanced CNN IP Restrictions section.
	  • Added support for *Resize operation* with scale factor of 2 only.
	• Updated the old *Caffe Restrictions* section.

Working with Projects	• Updated Figure 4.1. Project Implementation Options Window • Removed *Caffe* support in the Analyzed Network section. • Removed the old *Catdog* section. • Added YoloV8, YoloV11, and Resenet18v1 sections.

Advanced Topics	• Updated Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8, Figure 5.9, Figure 5.10, Figure 5.11, and Figure 5.12.
	• Removed *Caffe* support in the Data Preprocessing section.
	• Added the VE ALU Enable section.
	• Removed the old *Kmax Kernel Pooling* section.
	• Updated the LUT Input Bits and LUT Output Bits sections, to make this setting available across all ML IPs.
	• Updated the Learned Step Quantization (LSQ) section.
	  • Corrected the layer name from *Fully* to *Fully Connected*.
	  • Added *Lscquant.layers.QuantizeBatchNormalization* and *Lscquant.layers.Split* layer.
	• Added ONNX framework in Table 5.6. SensAI Security Flow: File Extension Mapping.

Supported Frameworks	Added support for *SoftMax*, and *Sigmoid* layers.
Model Zoo	Removed this section.
AI System Generator	Removed this section.
Appendix A	• Removed old *Appendix A. Supported and Added Caffe Layers* section and replaced it with *Supported Keras Layer*. • Added support for *GEMM*, *SiLU*, and *SoftMax* layers.

Section	Change Summary

Appendix B	• Updated Table B.1. Supported Layer Configuration.
	  • Added *BatchNorm* for all ML Ips.
	  • Added support for *SoftMax*, *SiLU* and *GEMM* for Advanced CNN IP.
	  • Updated *Global Average Pooling*, *Sigmoid*, and *Resize/Upsample* layers.
	  • Added *Unpooling*, *Split*, and *Concat* layers.
	  • Renamed *Resize/Bilinear* to *Resize/Upsample* layer.

Appendix C	• Updated the write up for *Matmul*. • Added support for *ResizeNearestNeighbour*, *Sigmoid*, *SoftMax*, and *SiLU* layers.
Appendix F	Added support for *YOLOv8*, *YOLOv11* and *ResNet18V1* topologies in Table F.1. Network Topology and Device.

Revision 7.0, December 2024

Section	Change Summary
All	• Updated Neural Network Compiler and Machine Learning Software version to 7.0. • Made minor editorial changes.
Abbreviations in This Document	• Updated section title, description, and table header. • Added *CLI*, *CSR*, *FC*, *FPQ*, *IP*, *LSQ*, *ML*, *NCHW*, *ONNX*, *ReLU*, *TCL*, and *USB*. • Rearranged items in alphabetical order.
Installing the Software	Updated, Figure 2.1. Installation Location Specification Figure 2.2. Installation Component Specification, and Figure 2.3. Installation Ready to Install Dialog Box.

Getting Started	• Added reference to *ONNX (experimental)* in relation to framework in the Creating a New Project section.
	• Added the Multiple Input Selection section under the Inputs section.
	• In Table 3.1. Arguments and Usage:
	  • Updated column header from *Programming Code* to *Argument*.
	  • Added ONNX to framework and network file arguments.
	  • Added *ip mode* argument.
	  • Updated argument names to *lut_input_bits {5,6,7,8,9,10,11,12}* and *lut_output_bits {8, 16}*.
	  • Added arguments *create_quantized_version {0, 1}*, *validation_data_path {path of directory}*, *enable_fc_4_bit_weights {0, 1}*, *number_of_ml_ips*, and *external_memory_port*.
	  • Added arguments for Multi-input Network.
	• In the CrossLink-NX and CertusPro-NX Optimized and Extended Mode Restrictions section:
	  • Added restrictions on 4-bit weights quantization, Focus Layer, and 4-bit activation in the Optimized IP mode.
	  • Added restriction on 4-bit input data to Fully Connected layer.
	• In the CertusPro-NX and Avant Advanced CNN IP Restrictions section:
	  • Added restriction on 4-bit activation in Advanced IP.
	• Added the ONNX Restrictions section.

Working with Projects	Added the Handgesture, MV1 (MobileNet V1), MV2 (MobileNet V2), YoloV5, and Toy_mnist sections.

Advanced Topics	• Updated Figure 5.11 Project Implementation Window – Avant Advanced IP Part 1.
	• In the Project Implementation Settings section:
	  • Added the Create Quantized Version, Validation Datapath, Enable FC 4 Bit Weight, Number of ML IPs, External Memory Port, and Initial LPDDR4 Address sections.
	• In the Quantization section:
	  • Added the Appendix B section.
	  • Reorganized content into the Fixed Point Quantization (FPQ) section.
	  • In the Fixed Point Quantization (FPQ) section:

Section	Change Summary

	• Re-organized content into and added description and code for Fixed Point Quantization Using Lscquant Package.
	• Updated title for Table 5.2. Unsigned 8-Bit Quantization (Fixed Point Quantization) and Table 5.3. Signed 8-Bit Quantization (Fixed Point Quantization).
	• Added 4b type in Table 5.4. Fixed Point Quantization Details with Device Type.
	• In Table 5.5 Quantization Support in Layers:
	  • Added ResizeBilinear.
	  • Updated quantization support description for *Convolution layer*, *MaxPooling or AveragePooling or ResizeBilinear*, *Batch norm layer*, and *Fully Connected layer*.
	• Added note on providing keras model .h5 as input if model is trained with LSQ.
	• Renamed sections starting from Fixed Point Quantization Training in Caffe through Fixed Point Quantization Requirements and Suggestions and updated descriptions.

Supported Frameworks	• Added reference to ONNX. • Removed reference to sigmoid as an unsupported data post-processing operation in the TensorFlow section. • Added the Using ONNX section.
AI System Generator	Added new section.
Appendix B. Supported Keras Layers	• Added sigmoid to supported Keras layers. • Added description for sigmoid and Figure B.1. Sigmoid Function.
Appendix C. Supported Layer Configuration	In Table B.1. Supported Layer Configuration: • Updated Optimized CNN and Advanced CNN values for the *Stride* parameter for the Convolution and Depthwise Convolution layers. • Added Advanced CNN value for the *Kernel*, *Stride*, and *Pad* parameters for the Global Average Pooling layer. • Added the sigmoid layer. • Updated Optimized CNN to supported for the Focus and Resize Bilinear layers.
Appendix F. Supported ONNX Layers	Added new section.
Appendix G. Network Topology and Device Table	Updated CrossLink-NX and CertusPro-NX support for Yolov5 to *Advanced and Optimized mode only* in Table F.1. Network Topology and Device.
References	Added Learned Step Size Quantization paper, Lattice sensAI Human Counting AI Demo webpage, USB3-Gigabit Ethernet Demo User Guide, and Lattice Semiconductor GitHub.

Revision 6.1, January 2024

Section	Change Summary
All	• Add support for YoloV5 models and layers like Conv 7x7, Mul, and Sub in Advanced IP. • Add the support of the 7x7 and 5x5 convolution kernels. • Add the support of the Global Average Pooling operation for FPQ network. • Add support for 64-bit datawidth in the Avant device Advanced IP. • Add the support of a strided slice and a focus layer. • Add the new Tensorflow native operations (Mul, Sub, Add, reciprocal_no_nan, Pow, Strided_Slice) as post-processing stand-alone nodes in Keras
Disclaimers	Updated this section.
Getting Started	Merged old subsection 3.6.1 Usage and subsection 3.6.2 Arguments into a new subsection 3.6.1 Arguments and Usage.
References	Add this section.

Revision 6.0, February 2023

Section	Change Summary
All	Added advanced IP support in CertusPro-NX and Advant devices.
Introduction	• Added Avant device support to the IP Requirements section. • Updated the description of downloading and running networks onto Hardware in the Purpose section.
Installing the Software	• Updated the default installation directory in Step 5. • Updated Figure 2 1. Installation Location Specification and Figure 2 3. Installation Ready to Install Dialog Box.
Getting Started	• Updated Arguments for new device family support in the Command Line Interface section. • Updated restrictions for new device family support in the CertusPro-NX and Avant Advanced CNN IP Restrictions section. • Newly added supported TenorFlow Version 2.9 in the TensorFlow Restrictions section.
Working with Projects	Newly added the HTML Log File section.

Advanced Topics	• Updated all the figures in this section reflecting the new GPO ID.
	• Newly added This option is available for Extended and Advanced CNN IP only to the
	• Argmax Memory Size section.
	• Added Avant device support to the following sections:
	  • On the Fly Post Processing
	  • Required Output Depth Range
	  • On the Fly Post Processing
	  • Required Output Depth Range
	  • On-Chip Large Memory Size
	  • External Memory Interfaced (In bytes)
	  • Code Section Base Address
	  • Data Section Base Address
	• Newly added the following sections:
	  • Number of Segments
	  • Segment Size
	  • Number of VE SPD
	  • Multiport Parallel
	  • Kmax Kernel Pooling
	• Added description about CertusPro-NX and Avant to the Number of Convolution Engines section.
	• Updated to it uses four DSP blocks per convolution engine in the Enable Quad Core Mode section.
	• Added This option is available for Extended and Advanced CNN IP only to the Argmax Memory Size section.
	• Added Avant device support to Table 5.3. Quantization Details with Device Type.
	• Added Avant device support to the Quantization for iCE40 UltraPlus, CrossLink NX, CertusPro NX, and Avant section.
	• Updated to Neural Network Compiler 6.0 in the Note in the Mobilenet Mode for ECP5 section.
	• Added except Advanced CNN IP for CertusPro-NX in the Embedded Mode section.

SB Debugging	Added Avant device support to the CNX VVML, CPNX section.
Technical Support Assistance	Added Lattice Answer Database URL.
Supported Keras Layers	Updated description for Lamboda (only for 8-bit activation quantization) section.
Supported Layer Configuration	Newly added the Advanced CNN column, Max Pooling K x K row, Argmax Pooling row of data to the table.
USB Debugging Register Map	Newly added the 0x0028, 0x0029, 0x002a, 0x002b, and 0x002c addresses.

Section	Change Summary
Network Topology and Device Table	Newly added Yolov5 network.
Common CNN Blocks Used in Lattice NNC	Newly added Appendix.

Revision 5.0, June 2022

Section	Change Summary
All	Added Extended IP, Semantic Segmentation Support, Updated USB Debug with enhancements

Revision 4.1, November 2021

Section	Change Summary
All	Added support for CertusPro-NX device and upgraded TensorFlow version support to 2.5.0.
	General editorial, style, and formatting update.

Revision 4.0, April 2021

Section	Change Summary
All	Added Concat and Large Input resolution support in CrossLink-NX device.

Revision 3.2, January 2020

Section	Change Summary
All	Added Quad LRAM support in CrossLink-NX device.

Revision 3.1, October 2020

Section	Change Summary
All	Added Mobilenet mode support for iCE40 UltraPlus device.

Revision 3.0, April 2020

Section	Change Summary
All	Added support for CrossLink-NX device.

Revision 2.1, September 2019

Section	Change Summary
All	Enhancements, bug fixes, and Mobilenet mode.

Revision 2.0, April 2019

Section	Change Summary
All	Added new features and optimizations.

Revision 1.1, September 2018

Section	Change Summary
All	Added support for iCE40 UltraPlus device.

Revision 1.0, May 2018

Section	Change Summary
All	Initial release.

www.latticesemi.com