CrossLink-NX QVGA MobileNet Human Identification Using VVML Board

Reference Design

FPGA-RD-02244-1.1

October 2023

Disclaimers

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

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
CNN	Convolutional Neural Network
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field-Programmable Gate Array
I^2C	Inter-Integrated Circuit
ML	Machine Learning
MLE	Machine Learning Engine
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
USB	Universal Serial Bus
VVML	Voice and Vision Machine Learning

1. Introduction

This reference design implements Convolutional Neural Network (CNN) based human face identification application on a low power Lattice FPGA using an image sensor. The training process is completed on a GPU-powered machine to sharpen the CNN to detect points of reference on a human face and measure them to distinguish the differences between people. This design can be used for identification of other objects by modifying the training database.

1.1. Design Process Overview

The design process involves the following steps:

1. Training the model
   - Setting up the basic environment
   - Preparing the dataset
   - Training the machine
     - Training the machine and creating the checkpoint data
   - Creating the frozen file (*. pb)
2. Compiling Neural Network
   - Creating the filter and firmware binary files with Lattice sensAI™ 4.1 program
3. FPGA design
   - Creating the FPGA Bitstream file
4. FPGA Bitstream and Quantized Weights and Instructions
   - Flashing the binary and bitstream files to CrossLink™-NX Voice and Vision Machine Learning (VVML) hardware

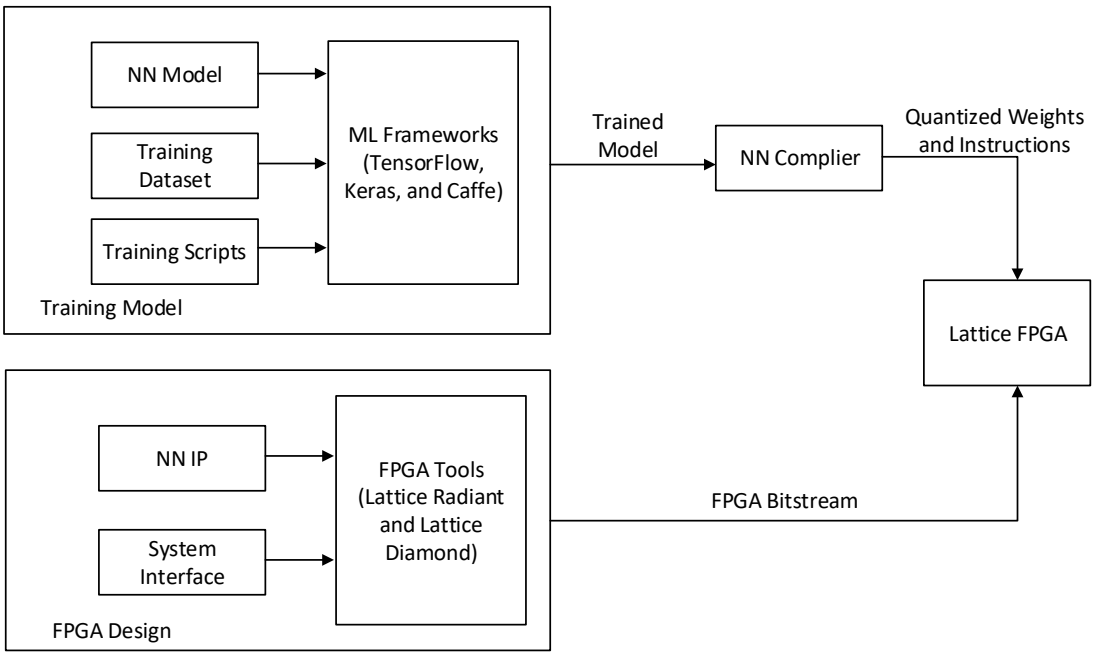

Figure 1.1. Lattice Machine Learning Design Flow

2. Setting Up the Basic Environment

2.1. Tools and Hardware Requirements

This section describes the required tools and environment setup for training and model freezing.

2.1.1. Lattice Tools

- Lattice Radiant™ Tool version 2.2 – Refer to http://www.latticesemi.com/latticeradiant
- Lattice Radiant Programmer version 3.0 – Refer to http://www.latticesemi.com/latticeradiant
- Lattice sensAI Compiler version 4.1 – Refer to https://www.latticesemi.com/Products/DesignSoftwareAndIP/AIML/NeuralNetworkCompiler

2.1.2. Hardware

This design uses the CrossLink-NX Voice and Vision board as shown in Figure 2.1.

- CrossLink-NX Voice and Vision Machine Learning (VVML) Board, Rev B Board.

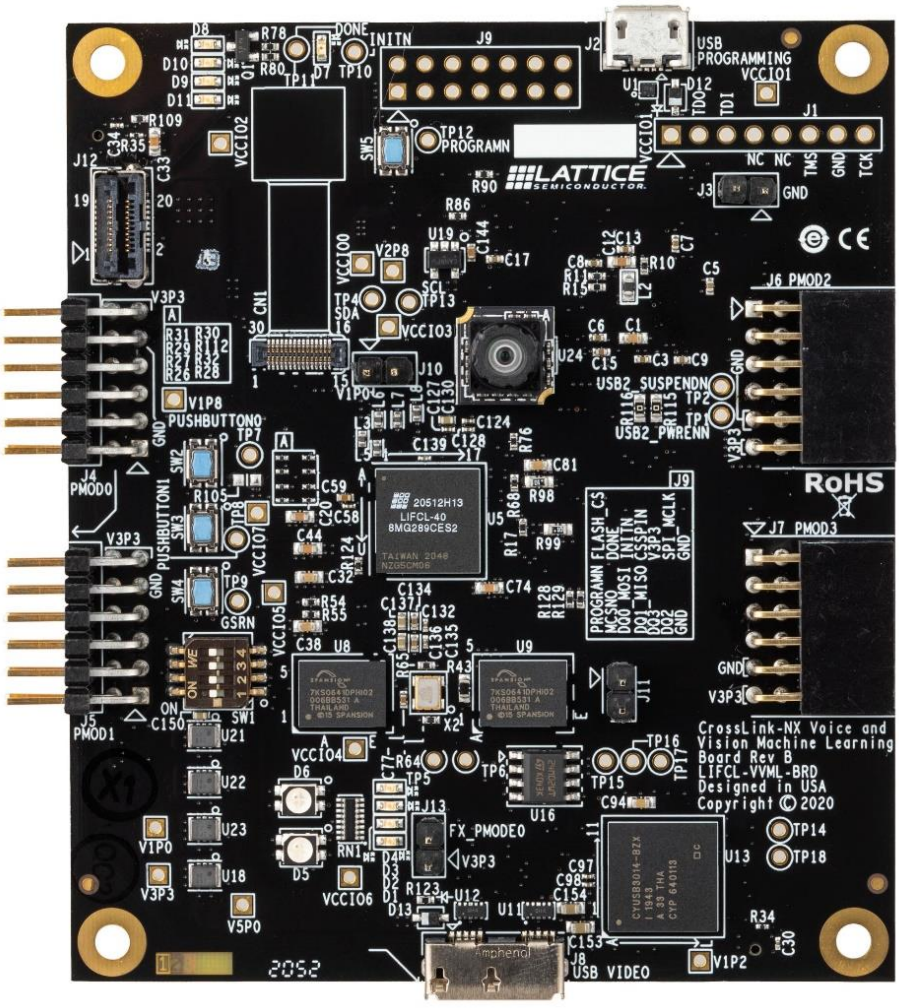

Figure 2.1. Lattice CrossLink-NX Voice and Vision Machine Learning (VVML) Board, Rev B

2.2. Setting Up the Linux Environment for Machine Training

This section describes the steps for NVIDIA GPU drivers and/or libraries for 64-bit Ubuntu 16.04 OS. NVIDIA library and TensorFlow version is dependent on PC and Ubuntu/Windows version.

2.2.1. Installing the NVIDIA CUDA and cuDNN Library for Machine Learning Training on GPU

2.2.1.1. Installing the CUDA Toolkit

To install the CUDA toolkit, run the following commands in the order specified below:

```
$ curl -O
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-
repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ curl -O https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2832  100  2832    0     0   2204      0  0:00:01  0:00:01 --:--:--  2205
```

Figure 2.2. Download CUDA Repo

```
$ sudo dpkg –I ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ sudo dpkg -i ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
Selecting previously unselected package cuda-repo-ubuntu1604.
(Reading database ... 5287 files and directories currently installed.)
Preparing to unpack .../cuda-repo-ubuntu1604_10.1.105-1_amd64.deb ...
Unpacking cuda-repo-ubuntu1604 (10.1.105-1) ...
Setting up cuda-repo-ubuntu1604 (10.1.105-1) ...

The public CUDA GPG key does not appear to be installed.
To install the key, run this command:
sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
```

Figure 2.3. Install CUDA Repo

```
$ sudo apt-key adv --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.
pub
```

```
$ sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
Executing: /tmp/tmp.a2QZZnTMUX/gpg.1.sh --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
gpg: key 7FA2AF80: public key "cudatools <cudatools@nvidia.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Figure 2.4. Fetch Keys

```
$sudo apt-get update
```

```
$ sudo apt-get update
Ign:1 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  InRelease
Hit:2 http://archive.ubuntu.com/ubuntu xenial InRelease
Get:3 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release [697 B]
Get:4 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release.gpg [836 B]
Hit:5 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:6 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:7 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
Ign:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages
Get:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages [428 kB]
Fetched 429 kB in 1s (386 kB/s)
Reading package lists... Done
```

Figure 2.5. Update Ubuntu Packages Repositories

```
$ sudo apt-get install cuda-9-0
```

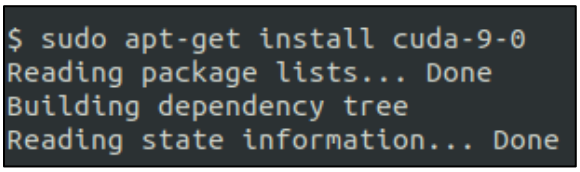

Figure 2.6. CUDA Installation

2.2.1.2. Installing the cuDNN

To install the cuDNN:

1. Create NVIDIA developer account: https://developer.nvidia.com.
2. Download cuDNN lib: https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.1.4/prod/9.0_20180516/cudnn-9.0-linux-x64-v7.1
3. Execute below commands to install cuDNN

```
$ tar xvfcudnn-9.0-linux-x64-v7.1.tgz
$ sudo cp cuda/include/cudnn.h/usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudochmoda+r/usr/local/cuda/include/cudnn.h/usr/local/cuda/lib64/libcudnn*
```

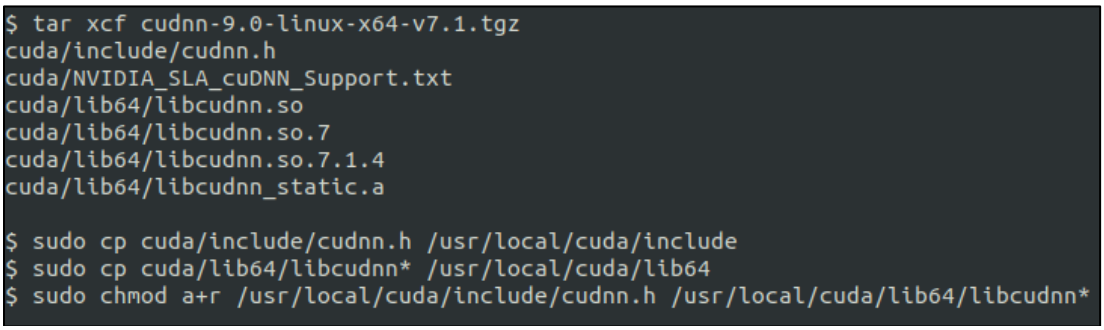

Figure 2.7. cuDNN Library Installation

2.2.2. Setting Up the Environment for Training and Model Freezing Scripts

This section describes the environment setup information for training and model freezing scripts for 64-bit Ubuntu 16.04. Anaconda provides one of the easiest ways to perform machine learning development and training on Linux.

2.2.2.1. Installing the Anaconda Python

To install the Anaconda and Python 3:

1. Go to https://www.anaconda.com/products/individual#download-section.
2. Download Python3 version of Anaconda for Linux.
3. Run the command below to install the Anaconda environment:

```
$  sh  Anaconda3-2019.03-Linux-x86_64.sh
```

Note: Anaconda3-**<version>**-Linux-x86_64.sh, version may vary based on the release.

```
(base) $ sh Anaconda3-2020.07-Linux-x86_64.sh

Welcome to Anaconda3 2020.07

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

Figure 2.8. Anaconda Installation

4. Accept the license.

```
Do you accept the license terms? [yes|no]
[no] >>> yes
```

Figure 2.9. Accept License Terms

5. Confirm the installation path. Follow the instruction on screen if you want to change the default path.

```
[no] >>> yes


Anaconda3 will now be installed into this location:
/home/user/anaconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/user/anaconda3] >>> /home/user/anaconda3
```

Figure 2.10. Confirm/Edit Installation Location

6. After installation, enter No as shown in Figure 2.11.

```
Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>  no
```

Figure 2.11. Launch/Initialize Anaconda Environment on Installation Completion

2.2.3. Installing the TensorFlow version 2.1.0

To install the TensorFlow version 2.1.0:

1. Activate the conda environment by running the command below:

```
$ source <conda directory>/bin/activate
```

```
$ source anaconda3/bin/activate
(base) ~$
```

Figure 2.12. Anaconda Environment Activation

2. Install the TensorFlow by running the command below:

```
$ conda install tensorflow-gpu==2.1.0
```

```
$ conda install tensorflow-gpu=2.1.0
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: [redacted]

  added / updated specs:
    - tensorflow-gpu=2.1.0
```

Figure 2.13. TensorFlow Installation

3. After installation, enter **Y** as shown in Figure 2.14.

```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    aiohttp-3.7.4.post0        |   py37h7f8727e_2         541 KB
    async-timeout-3.0.1        |   py37h06a4308_0          13 KB
    cffi-1.14.6                |   py37h400218f_0         223 KB
    cryptography-3.4.7         |   py37hd23ed53_0         904 KB
    multidict-5.1.0            |   py37h27cfd23_2          66 KB
    pyjwt-2.1.0                |   py37h06a4308_0          32 KB
    tensorflow-2.1.0           |gpu_py37h7a4bb67_0           4 KB
    tensorflow-base-2.1.0      |gpu_py37h6c5654b_0       155.3 MB
    tensorflow-gpu-2.1.0       |       h0d30ee6_0           3 KB
    ------------------------------------------------------------
                                           Total:       157.0 MB
```

Figure 2.14. TensorFlow Installation Confirmation

Figure 2.15 shows TensorFlow installation is complete.

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Figure 2.15. TensorFlow Installation Completion

2.2.4. Installing the Python Package

To install the Python package:

1. Install Easydict by running the command below:

```
$ conda install -c conda-forge easydict
```

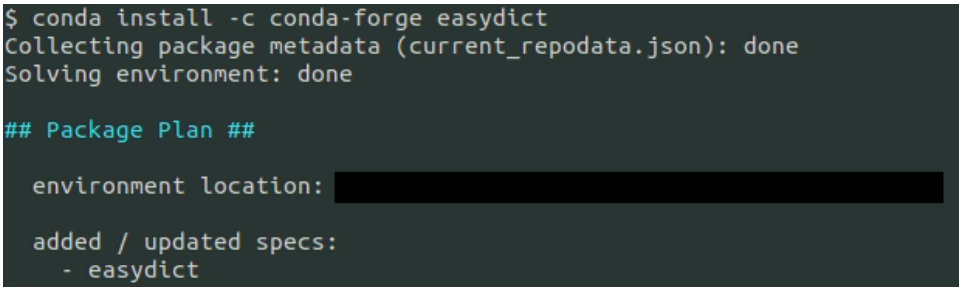

Figure 2.16. Easydict Installation

2. Install opencv by running the command below:

```
$ conda install -c menpo opencv
```

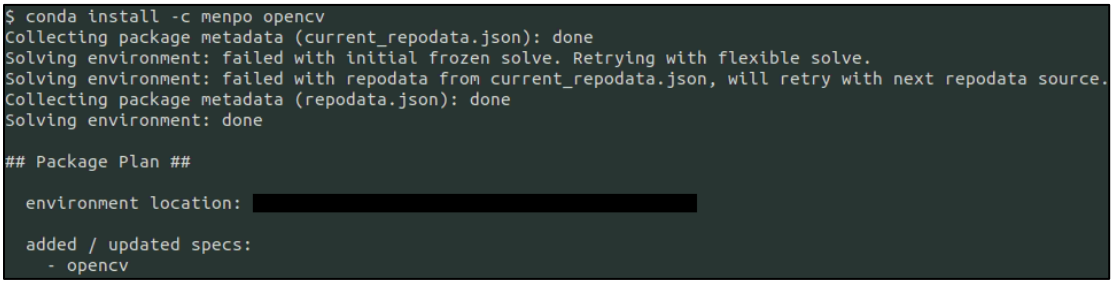

Figure 2.17. Opencv Installation

3. Install bcolz by running the command below:

```
$ conda install bcolz
```

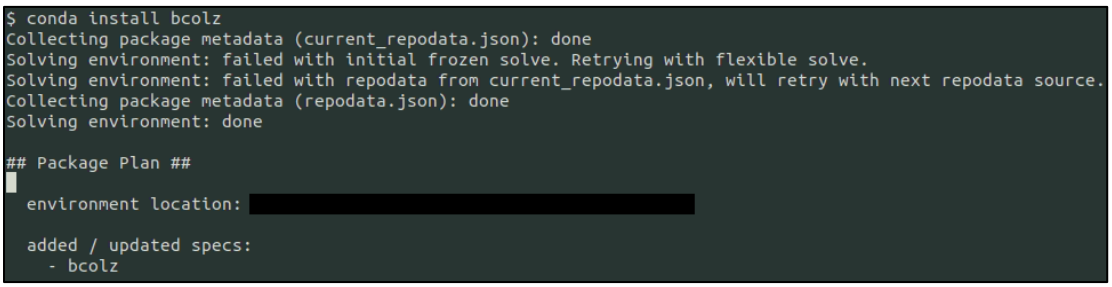

Figure 2.18. Bcolz Installation

4. Install tensorflow-addons by running the command below:

```
$ python -m pip install tensorflow-addons==0.9.1
```

```
$ python -m pip install tensorflow-addons==0.9.1
Collecting tensorflow-addons==0.9.1
  Downloading tensorflow_addons-0.9.1-cp37-cp37m-manylinux2010_x86_64.whl (1.0 MB)
     |                                | 1.0 MB 2.4 MB/s
Installing collected packages: tensorflow-addons
  Attempting uninstall: tensorflow-addons
    Found existing installation: tensorflow-addons 0.14.0
    Uninstalling tensorflow-addons-0.14.0:
      Successfully uninstalled tensorflow-addons-0.14.0
Successfully installed tensorflow-addons-0.9.1
```

Figure 2.19. Tensorflow-addons Installation

5. Install scikit-learn by running the command below:

```
$ conda install scikit-learn
```

```
$ conda install scikit-learn
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: [redacted]

  added / updated specs:
    - scikit-learn
```

Figure 2.20. Scikit-learn Installation

6. Install tqdm by running the command below:

```
$ conda install tqdm
```

```
$ conda install tqdm
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: [redacted]

  added / updated specs:
    - tqdm
```

Figure 2.21. Tqdm Installation

7. Install albumentations by running the command below:

```
$ pip install albumentations
```

```
(base) $ pip install albumentations
Collecting albumentations
  Downloading albumentations-1.0.3-py3-none-any.whl (98 kB)
     |                                | 98 kB 557 kB/s
Collecting opencv-python-headless>=4.1.1
  Downloading opencv_python_headless-4.5.3.56-cp38-cp38-manylinux2014_x86_64.whl (37.1 MB)
Installing collected packages: opencv-python-headless, albumentations
Successfully installed albumentations-1.0.3 opencv-python-headless-4.5.3.56
```

Figure 2.22. Albumentations Installation

3. Dataset Preparation

This section describes the steps and guidelines used to prepare the dataset to train the Face ID Demo for CrossLink-NX Voice and Vision Machine Learning (VVML) board. Note that this section is for the example reference. The following sections provide the guidelines and/or example which can be used as reference for preparing dataset for given use cases but in no case, Lattice is recommending and/or endorsing any dataset(s). Lattice strongly recommends customers to gather and prepare their own datasets for their end applications.

3.1. Dataset Information

For Face ID, use the dataset in directory format used in the image classification problem. Each folder in dataset denotes individual class which contains multiple images of that class.

Figure 3.1. Dataset Format

3.1.1. Training Set

- For the Face ID demo, use the MS-Celebs-1m dataset – https://www.microsoft.com/en-us/research/project/ms-celeb-1m-challenge-recognizing-one-million-celebrities-real-world/ and https://github.com/EB-Dodo/C-MS-Celeb
- The original dataset does not have aligned faces. You can download the face aligned dataset in https://drive.google.com/file/d/1X202mvYe5tiXFhOx82z4rPiPogXD435i/view, or align faces using the custom scripts and cv2 operations.

3.1.2. Validation set

- The subset of MS-celebs dataset is used as validation set which has 64 identities, with total of 4472 images.

3.1.3. Test Set

- As test set, use the subset from the LFW Face dataset (http://vis-www.cs.umass.edu/lfw/).
- The test set contains total of 6000 pairs of images, where 3000 pairs are of same person and another 3000 pairs of different persons.

4. Training Code Preparation

4.1. Training Code Structure

Download the Lattice Face ID demo training code. Figure 4.1 shows the directory structure of the Face ID demo:

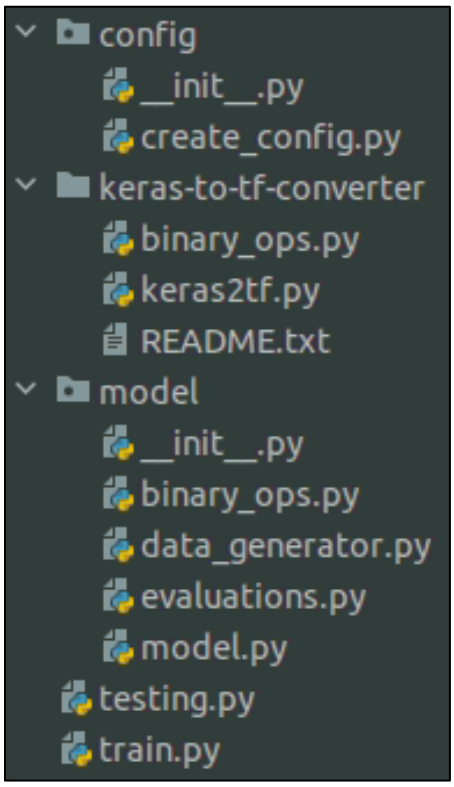

Figure 4.1. Training Code Directory Structure

4.2. Neural Network Architecture

4.2.1. Neural Network Architecture

This section provides information on the Convolution Neural Network configuration of the Face Identification design.

Table 4.1. Face Identification Training Network Topology

Image Input (112 × 112 × 1)		
Fire 1	DWConv3 – 40	Conv3 - # where: • Conv3 = 3 × 3 Convolution filter Kernel size • # = The number of filter DWConv3 – 32- # where: • DWConv3 = Depth wise convolution filter with 3 × 3 size • # = The number of filter Conv1 – 32- # where: • Conv1 = 1 × 1 Convolution filter Kernel size • # = The number of filter For example, Conv3 – 16 = 16 3 × 3 convolution filters BN – Batch Normalization
	BN	
	ReLU	
	Maxpool	
	Conv1 – 40	
	BN	
	ReLU	
Fire 2	DWConv3 – 40	
	BN	
	ReLU	
	Conv1 – 40	
	BN	
	ReLU	
Fire 3	DWConv3 – 60	
	BN	
	ReLU	
	Maxpool	
	Conv1 – 60	
	BN	
	ReLU	
Fire 4	DWConv3 – 60	
	BN	
	ReLU	
	Conv1 – 60	
	BN	
	ReLU	
Fire 5	DWConv3 – 80	
	BN	
	ReLU	
	Maxpool	
	Conv1 – 80	
	BN	
	ReLU	
Fire 6	DWConv3 – 80	
	BN	
	ReLU	
	Conv1 – 80	
	BN	
	ReLU	

Image Input (112 × 112 × 1)		
Fire 7	DWConv3 – 100	
	BN	
	ReLU	
	Maxpool	
	Conv1 – 100	
	BN	
	ReLU	
Fire 8	DWConv3 – 40	
	BN	
	ReLU	
	Conv1 – 40	
	BN	
	ReLU	
Dropout	Dropout – 0.3	
E_Dense	Dense – 128	

- In Table 4.1, the layer contains convolution (Conv), batch normalization (BN), ReLU, and pooling layers.
- Layer information
  - Convolutional Layer
    In general, the first layer in a CNN is always a convolutional layer. Each layer consists of number of filters (sometimes referred as kernels) which convolves with input layer/image and generates activation map (that is feature map). This filter is an array of numbers (the numbers are called weights or parameters). Each of these filters can be thought of as feature identifiers, like straight edges, simple colors, curves, and other high-level features. For example, the filters on the first layer convolve around the input image and “activate” (or compute high values) when the specific feature (say curve) it is looking for is in the input volume.
  - ReLU (Activation layer)
    It is the convention to apply a nonlinear layer (or activation layer) immediately after each conv layer. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference in accuracy. The ReLU layer applies the function f(x) = max (0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0.This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.
  - Pooling Layer
    After some ReLU layers, you may choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are also several layer options, with Maxpooling being the most popular. This basically takes a filter (normally of size 2 × 2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around.
    The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there is a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost.

    The second is that it controls over fitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of over fitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

- Batch Normalization Layer
  Batch normalization layer reduces the internal covariance shift. To train a neural network, some preprocessing to the input data are performed. For example, you could normalize all data so that it resembles a normal distribution (that means, zero mean and a unitary variance). This prevents the early saturation of non-linear activation functions such as sigmoid, assures that all input data is in the same range of values, and others.
  An issue, however, appears in the intermediate layers because the distribution of the activations is constantly changing during training. This slows down the training process because each layer must learn to adapt them to a new distribution in every training step. This is known as internal covariate shift.
  Batch normalization layer forces the input of every layer to have approximately the same distribution in every training step by following the process below during training:
  a. Calculate the mean and variance of the layers input.
  b. Normalize the layer inputs using the previously calculated batch statistics.
  c. Scales and shifts in order to obtain the output of the layer.
  This makes the learning of layers in the network more independent of each other and allows you to be care-free about weight initialization, works as regularization in place of dropout and other regularization techniques.
- Drop-out layer
  Dropout layers have a very specific function in neural networks. After training, the weights of the network are so tuned to the training examples they are given that the network do not perform well when given new examples. The idea of dropout is simplistic in nature. This layer *drops out* a random set of activations in that layer by setting them to zero. It forces the network to be redundant. That means the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too fitted to the training data and thus helps alleviate the over fitting problem. An important note is that this layer is only used during training, and not during test time.
- Fully connected layer
  This layer basically takes an input volume (whatever the output is of the conv or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program must choose from.
- Quantization
  Quantization is a method to bring the neural network to a reasonable size, while also achieving high performance accuracy. This is especially important for on-device applications, where the memory size and number of computations are necessarily limited. Quantization for deep learning is the process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

The architecture above provides nonlinearities and preservation of dimension that help to improve the robustness of the network and control over fitting.

4.2.2. Face Identification Network Output

The Face Identification model gives an output of 128 embeddings.

4.2.3. Training Code Overview

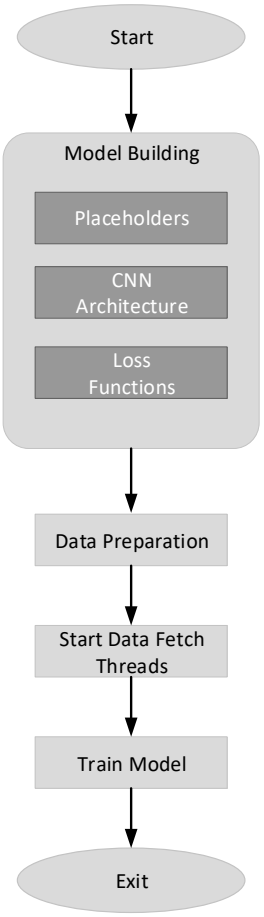

Figure 4.2. Training Code Flow Diagram

Training code can be divided into below parts:

- Model Configuration
- Model Building
- Data Preparation
- Training for overall execution flow

The details of each part can be found in the subsequent sections.

4.2.3.1. Model Configuration

The following is a summary of the configurable parameters in the *config/create_config.py* file:

- Dataset Configuration

```
cfg.dataset_path = "FaceID-datasets/ms-celebs-1m/"
cfg.validation_set = "FaceID-datasets/validation-set/"
cfg.test_set = "FaceID-datasets/test-set/"
```

Figure 4.3. Code Snippet – Input Dataset Path Configuration

- Model Properties

```
cfg.MODEL_NAME = 'Face_Identification'
cfg.LOG_PATH = "logs"
```

Figure 4.4. Code Snippet – Model Name and Log Directory

- Image Size
  - Change *cfg.IMAGE_WIDTH* and *cfg.IMAGE_HEIGHT* to configure Image size (width and height) if required.

```
cfg.IMAGE_WIDTH = 112
cfg.IMAGE_HEIGHT = 112
cfg.N_CHANNELS = 1
```

Figure 4.5. Code Snippet – Input Image Size Configuration

- Model Configuration

```
cfg.FILTER_DEPTHS = [40, 40, 60, 60, 80, 80, 100, 40]
cfg.EARLY_POOLING = True
cfg.USE_CONV3 = True
```

Figure 4.6. Code Snippet – Model Features and Depth Configuration

```
cfg.EPOCHS = 120
cfg.QUANT_RANGE = (0, 2)

cfg.FEATURES = 128
cfg.BATCHSIZE = 256

cfg.GPUID = 0
cfg.LEARNING_RATE = 0.01
cfg.REDUCELRONPLATEAU = True

cfg.TEST_BATCHSIZE = 64
cfg.IS_CROP = False
```

Figure 4.7. Code Snippet – Training Parameters

  - You can configure the number of layers by adding and removing the number of depth in FILTER_DEPTHS.
  - FEATURES denote how many values you need at embedding the layer.
  - If you set S_CROP to true, images are zoomed in and used for training.
  - REDUCELRONPLATEAU configures auto-reducing learning rate if loss is not reducing at some point.

4.2.3.2. Model Building

SqueezeDet class constructor builds model, which is divided into the following sections:

Forward Graph

- The CNN architecture consists of Convolution, Batch Normalization, ReLU, and Maxpool layers.
- Forward graph consists of eight fire layers as described in Table 4.1.
- You can set the pooling structure in *models/model.py* as per the depth you selected.

```
if config.EARLY_POOLING:
    if depth_length == 6:
        self.pooling = [True, True, True, False, True, False]
    elif depth_length == 7:
        self.pooling = [True, False, True, False, True, False, True]
    elif depth_length == 8:
        self.pooling = [True, False, True, False, True, False, True, False]
    elif depth_length == 9:
        self.pooling = [True, False, True, True, False, False, True, False, False]
    elif depth_length == 12:
        self.pooling = [True, False, False, True, False, True, False, True, False, False, False, False]
else:
    if depth_length == 6:
        self.pooling = [True, False, True, True, True, False]
    elif depth_length == 7:
        self.pooling = [True, False, True, False, True, False, True]
    elif depth_length == 8:
        self.pooling = [True, False, True, False, True, False, True, False]
    elif depth_length == 9:
        self.pooling = [True, False, True, False, True, False, True, False, False]
    else:
        self.pooling = [True, False, False, True, False, True, False, True, False, False]
```

Figure 4.8. Code Snippet – Pooling Structure

- You can mark the layer index *True* if you want pooling in that index.

```
self.input_layer = Input(shape=(self.config.IMAGE_HEIGHT, self.config.IMAGE_WIDTH, self.config.N_CHANNELS
                         name="input")
prev_layer = self.input_layer
if self.useconv3:
    self.fire1 = self._fire_layer(name="fire1", input=prev_layer, out_channels=self.depth[0], stdd=0.012,
                                  wt_quant=True, act_quant=True, bn_eps=1e-3, pool=self.pooling[0])
else:
    self.fire1 = self._mobile_layer(name="fire1", input=prev_layer, out_channels=self.depth[0], stdd=0.012,
                                    wt_quant=True, act_quant=True, bn_eps=1e-3, pool=self.pooling[0],
                                    pool2=True)
prev_layer = self.fire1
for layer_count in range(len(self.depth) - 1):
    if self.config.BACK_BONE == "VGG":
        prev_layer = self._fire_layer(name="fire{}".format(str(layer_count + 2)), input=prev_layer,
                                      out_channels=self.depth[layer_count + 1],
                                      stdd=0.012, wt_quant=True, act_quant=True,
                                      bn_eps=1e-3, pool=self.pooling[layer_count + 1])
    elif self.config.BACK_BONE == "MV1":
        prev_layer = self._mobile_layer(name="fire{}".format(str(layer_count + 2)), input=prev_layer,
                                        out_channels=self.depth[layer_count + 1],
                                        stdd=0.012, wt_quant=True, act_quant=True,
                                        bn_eps=1e-3, pool=self.pooling[layer_count + 1])
    else:
        print("Invalid Backbone !")
        sys.exit()
```

Figure 4.9. Code Snippet – Forward Graph Fire Layer

- Figure 4.10 shows the generated model using the model configuration you provided in the *config* file.

```
if 0 < dropout < 1:
    nn = Dropout(dropout)(prev_layer)
else:
    nn = prev_layer
nn = Flatten(name="E_flatten")(nn)
nn = Dense(self.config.FEATURES, use_bias=use_bias, kernel_initializer="glorot_normal",
           name="E_dense", kernel_constraint=MinMaxNorm(
        min_value=-0.5, max_value=0.5, rate=1.0, axis=0))(nn)

nn = tf.keras.layers.Lambda(lambda x: tf.math.l2_normalize(x, axis=1))(nn)
```

Figure 4.10. Code Snippet – Forward Graph Output Layer

- The output layer has dropout, followed by the dense layer with the number of features you set in the *config* file. Note that the Dense layer is normalized by L2 normalization while training and it will be removed while freezing.
- The L2 Normalization calculates the distance of the vector coordinate from the origin of the vector space. The result is positive distance value.

Loss Graph and Optimizer

```
self.face_rec_model.compile(optimizer=Adam(learning_rate=self.cfg.LEARNING_RATE),
                            loss=tfa.losses.TripletSemiHardLoss())
```

Figure 4.11. Code Snippet – Forward Graph Triplet Loss

- The Tensorflow's *TrippletSemiHardLoss* is used as loss to normalize embeddings.
- This block is responsible for training the model with the Adam optimizer to reduce all losses.

4.2.3.3. Training

```
def train_model(self):
    self.check_for_existing_checkpoints()
    self.face_rec_model.fit_generator(self.tripplet_data_generator, epochs=self.cfg.EPOCHS,
                                      validation_data=(self.val_data_generator),
                                      verbose=1, initial_epoch=self.init_epoch, callbacks=self.callbacks)
```

Figure 4.12. Code Snippet: Training

The Fit-generator feeds the data and labels batches to the Keras network and optimizes the weights and biases.

4.3. Training from Scratch and/or Transfer Learning

To train the machine:

1. Go to the top/root directory of the Lattice training code from command prompt.

   The Model works on 112 × 112 images.

   Current Face ID training code uses mean = 0 and scale = 1/128 (0.0078125) in pre-processing step with grayscale images. Mean and scale can be changed in training code *@src/dataset/imdb.py* as shown in Figure 4.13.

```
img = cv2.imread(self.paths[ID])
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.resize(img, self.dim)
img = self.__augment(img)
img = img[:, :, None]
img = img / 128.
```

Figure 4.13. Training Code Snippet for Mean and Scale

   The triplet data generator trains 10k identities on one epoch at a time and it shuffles the ids on each epoch end to train with next 10k random identities.

```
def on_epoch_end(self):
    """Updates indexes after each epoch"""
    import random
    random.shuffle(self.paths)
    for path in self.paths[:self.num_person_per_epoch]:
        self.paths1.extend(glob(path + "**"))
    self.paths = self.paths1
    self.classes = np.unique(np.asarray([path.split('/')[-2] for path in self.paths]))
    self.index_mapping = {}
    for i, class_id in enumerate(self.classes):
        self.index_mapping[class_id] = i
    self.n_classes = len(self.classes)
    self.indexes = np.arange(len(self.paths))
    if self.shuffle:
        np.random.shuffle(self.indexes)
    self.__len__()
```

Figure 4.14. Training Code Snippet for Identity Shuffling on Epoch End

The data generator performs on the Fly augmentation using the *Albumentation* library. You are basically performing contrast, brightness, and rotation (+-15') for augmentations.

```
self.transform = A.Compose([
    A.RandomBrightnessContrast(0.2, 0.2, False, p=0.2),
    A.Rotate(limit=(-15,15), interpolation=1, border_mode=cv2.BORDER_CONSTANT, value=0,
             mask_value=0, always_apply=False, p=0.3)

])
```

Figure 4.15. Training Code Snippet for On the Fly Augmentation

2. Execute the *train.py* file to start training.

```
$ python train.py
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           [(None, 112, 112, 1)]     0
_________________________________________________________________
fire1/expand3x3 (Conv2D)     (None, 112, 112, 40)      360
_________________________________________________________________
batch_normalization (BatchNo (None, 112, 112, 40)      160
_________________________________________________________________

|| || || || || || || || || || || || || || || || || || || || ||
_________________________________________________________________
dropout (Dropout)            (None, 7, 7, 40)          0
_________________________________________________________________
E_flatten (Flatten)          (None, 1960)              0
_________________________________________________________________
E_dense (Dense)              (None, 128)               251008
_________________________________________________________________
lambda_8 (Lambda)            (None, 128)               0
=================================================================
Total params: 530,568
Trainable params: 529,568
Non-trainable params: 1,000
_________________________________________________________________
None
Total  692715  with  10000  number of classes found.
Total  8944  with  64  number of classes found.
Train for 1352 steps, validate for 17 steps
Epoch 1/120
1351/1352 [============================>.] - ETA: 0s - loss: 0.9963
Epoch 00001: loss improved from inf to 0.99629, saving model to logs/checkpoints/model.01-1.00.hdf5
1352/1352 [==============================] - 450s 333ms/step - loss: 0.9963 - val_loss: 0.9978
```

Figure 4.16. Execute Run Script

3. Start TensorBoard.
   ```
   $ tensorboard –logdir=<log directory of training>
   ```
   For example: *tensorboard –logdir='./logs/'*.
4. Open the local host port on your web browser.

```
$ tensorboard --logdir logs/tensorboard/train/
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.4.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

Figure 4.17. TensorBoard – Generated Link

5. Check the training status on TensorBoard.

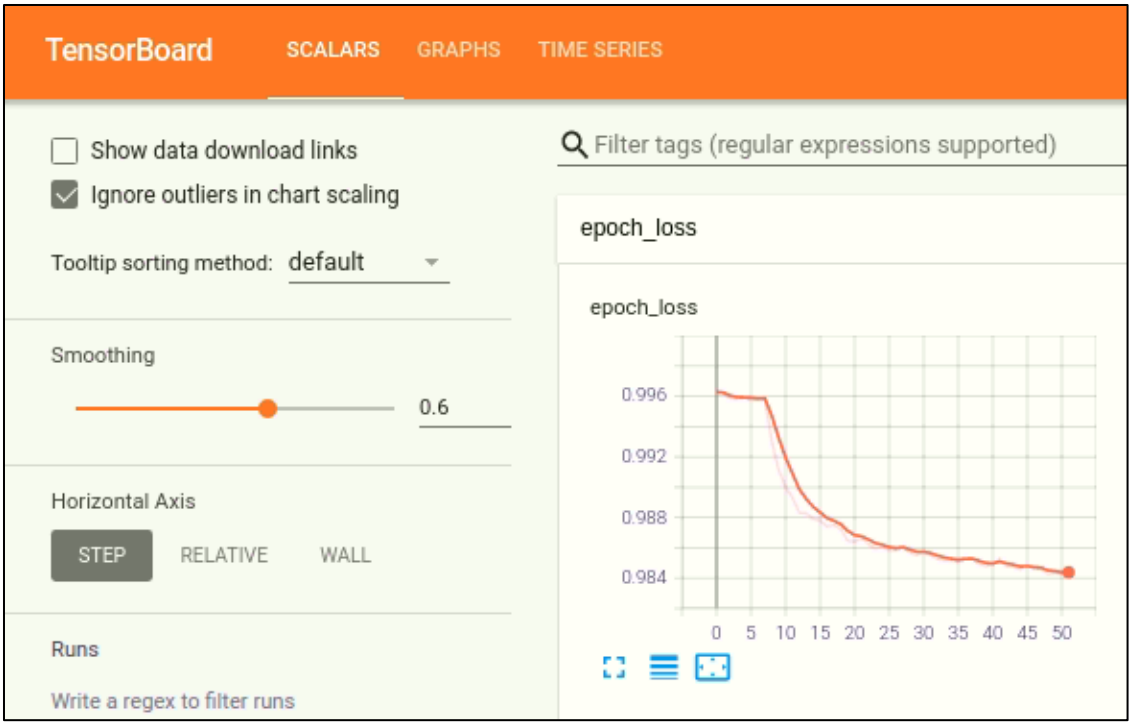

Figure 4.18. TensorBoard

Figure 4.19 shows the image menu of TensorBoard.

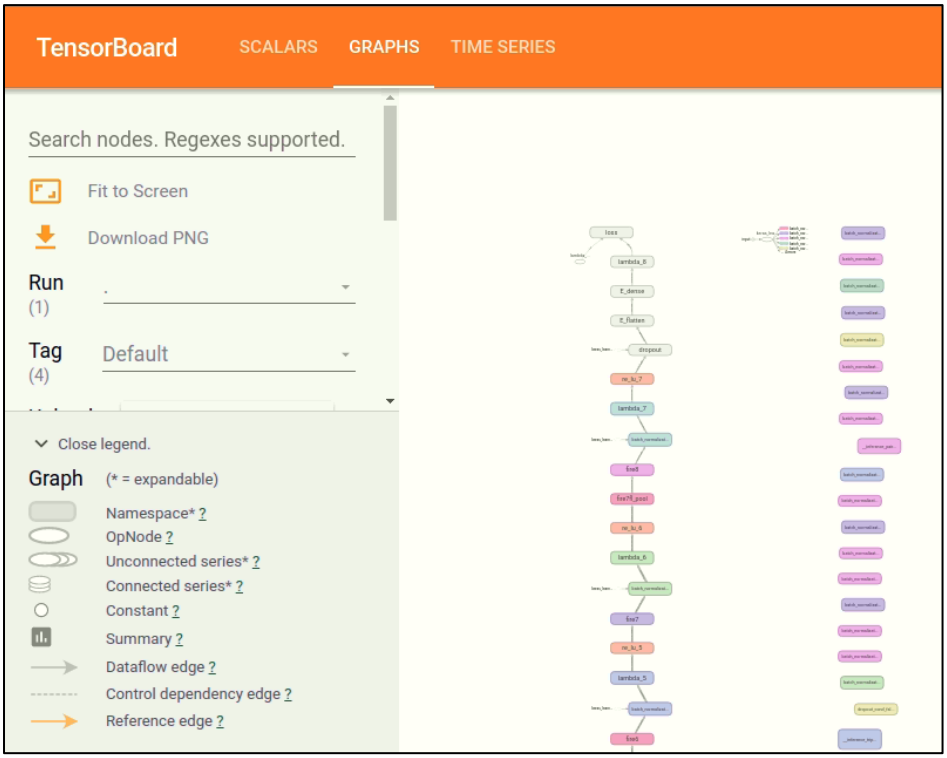

Figure 4.19. Model Graph on TensorBoard

6. Checkpoints are saved at the end of each epoch. At the end of training, code also saves the frozen graph with original graph as shown in Figure 4.20.

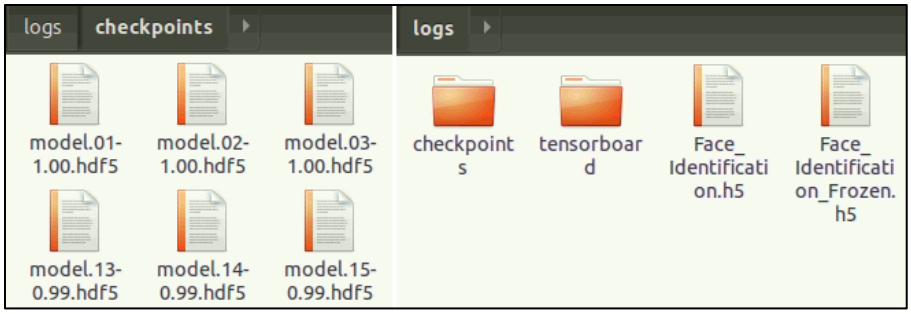

Figure 4.20. Example of Checkpoint Data Files at Log Folder

7. If you start training with the existing log file, the training resumes from the latest checkpoint. Alternatively, you can specify the pretrained model in the *config* file. The model restores the weights from the pretrained model.

   Note that while using the pretrained model for transfer learning, the learning-rate should be set low.

```
cfg.IMAGE_WIDTH = 112
cfg.IMAGE_HEIGHT = 112
cfg.N_CHANNELS = 1

cfg.init = None  # "<Pretrained H5 file path>"

cfg.FILTER_DEPTHS = [40, 40, 60, 60, 80, 80, 100, 40]
```

Figure 4.21. Set Pretrained Model Path to init Variable for Transfer Learning

5. Evaluating the Model

This section describes the procedure on how to calculate the model performance in terms of correct detection percentage.

5.1. Running Inference on Test Set

The Face Identification code contains the *testing.py* script under root directory. It takes the input model and uses the configuration created for training to run the testing on the test set.

```
$ python testing –model <Model Path>
```

Figure 5.1. Run Testing

The testing results also suggest optimal threshold for deployment.

6. Creating Binary File with Lattice sensAI

This chapter describes how to generate the binary file using the Lattice sensAI version 4.1 program.

Figure 6.1. sensAI – Home Screen

To create the project in the sensAI tool:

1. Click **File > New**.
2. Enter the following settings:
   - **Project Name**
   - **Framework – Keras**
   - **Class – CNN**
   - **Device – CrossLink-NX**
3. Click **Network File** and select the network (h5) file.

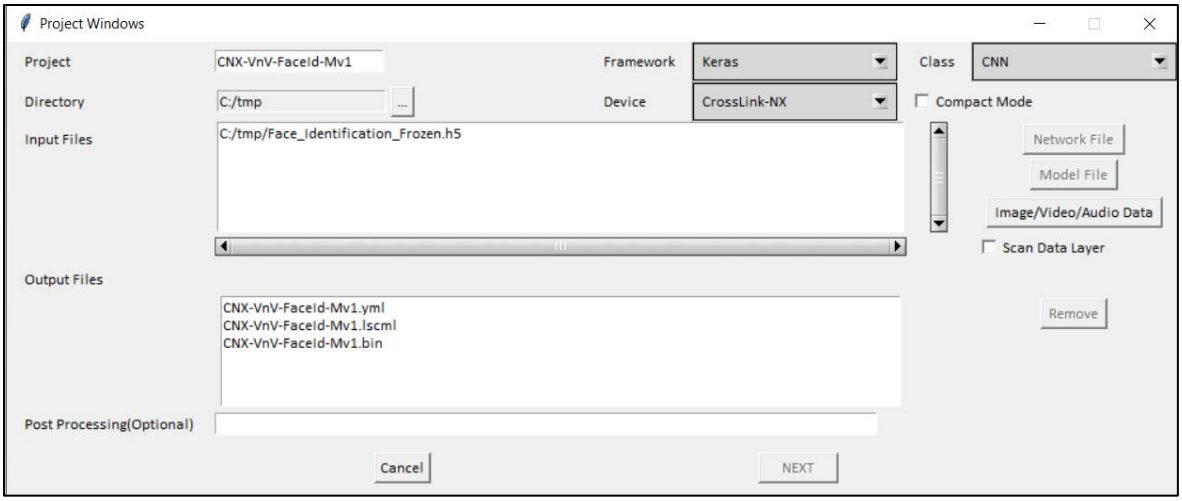

Figure 6.2. sensAI – Framework, Device, and Network File Selection

4. Click **Image/Video/Audio Data** and select the image input file.

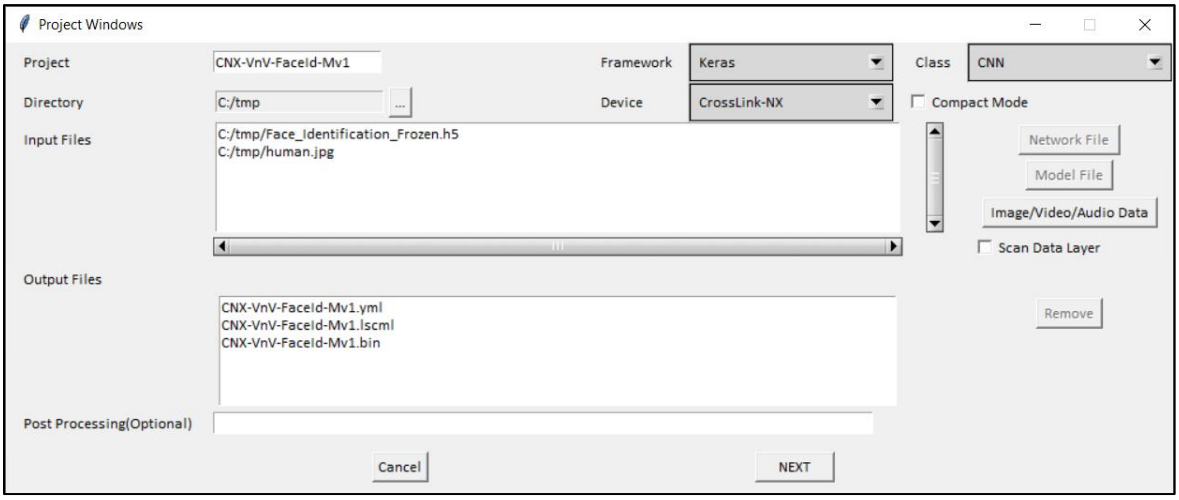

Figure 6.3. sensAI – Image Data File Selection

5. Click **Next**.
6. Configure your project settings as shown in Figure 6.4.
   - Mean Value for Data Pre-Processing – **0**
   - Scratch Pad Memory Block Size – **8192**
   - On-Chip Memory Block Size – **131072**
   - Scale Value for Data Pre-Processing – **0.0078125**
   - Data Selection Base Address – **4194304**

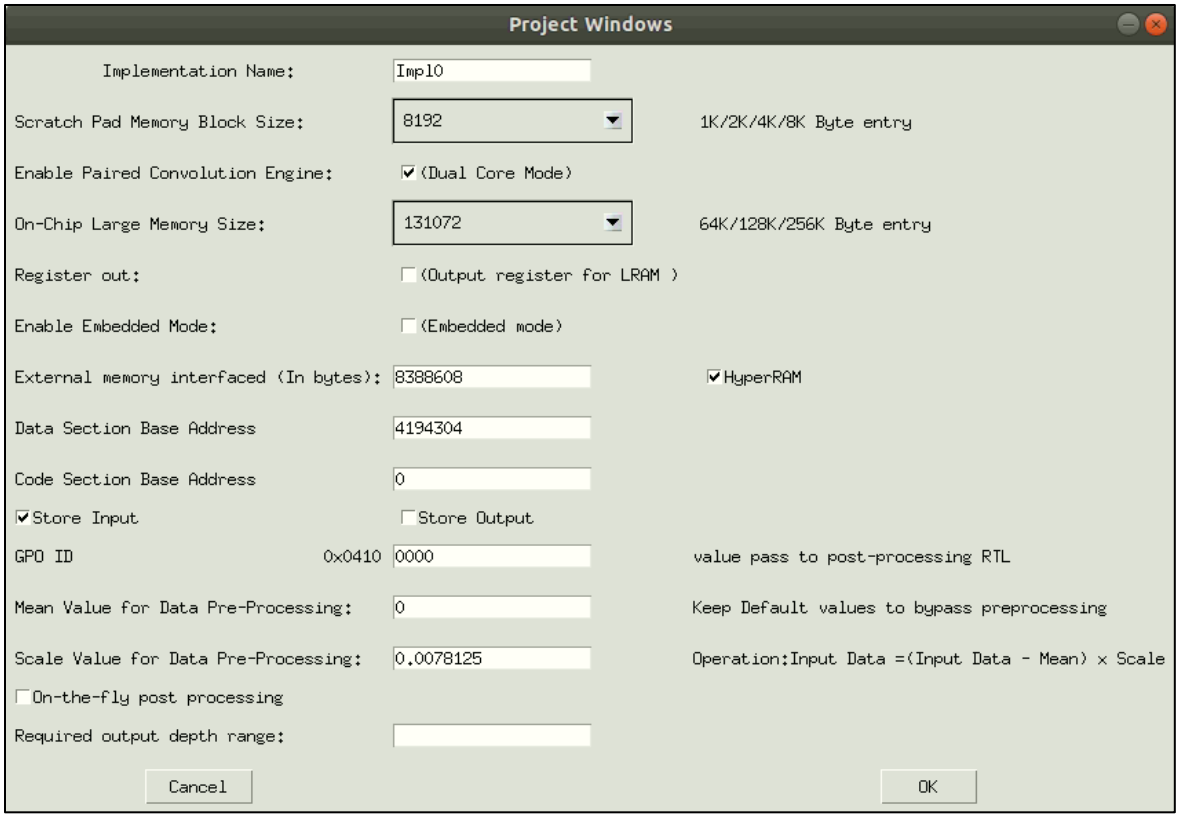

Figure 6.4. sensAI – Update Project Settings

7. Click **OK** to create the project.
8. Double-click **Analyze**.

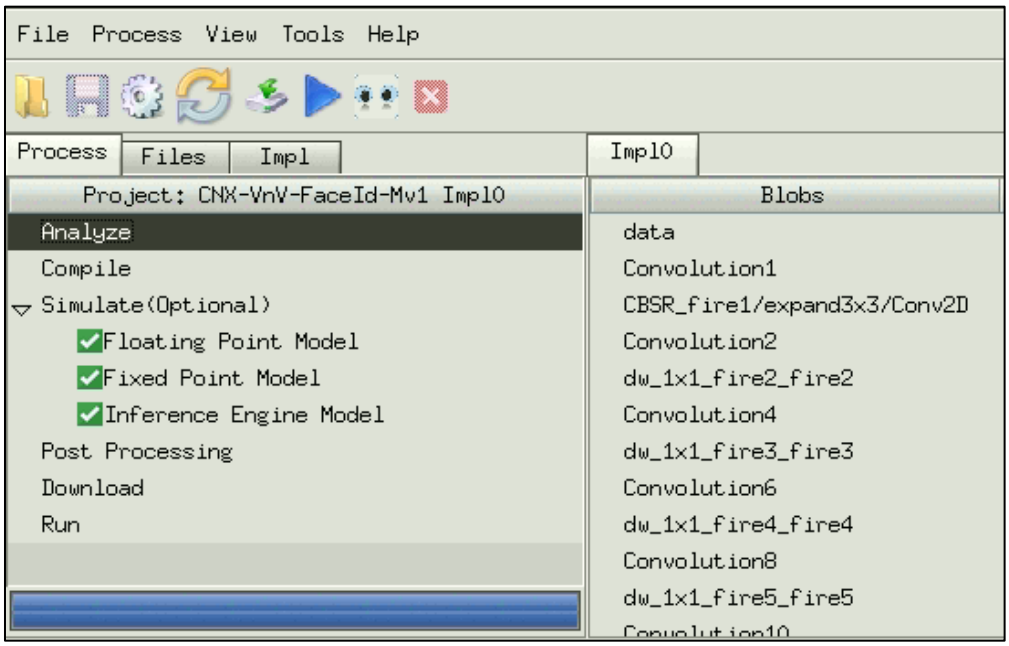

Figure 6.5. Analyze Project

9. Confirm the Q format of each layer as shown in Figure 6.6.

Impl0

Blobs	Data Format (Analyzed)	Stored Data Format	Internal(External) Mem Byte
data	8.7	1.7	12544 (12544)
Convolution1	5.10	5.10	None
CBSR_fire1/expand3x3/Conv2D	8.7	1.7	53760 (125440)
Convolution2	5.10	5.10	None
dw_1x1_fire2_fire2	8.7	1.7	60480 (206080)
Convolution4	5.10	5.10	None
dw_1x1_fire3_fire3	8.7	1.7	25440 (87360)
Convolution6	5.10	5.10	None
dw_1x1_fire4_fire4	8.7	1.7	25088 (N/A)
Convolution8	5.10	5.10	None
dw_1x1_fire5_fire5	8.7	1.7	7840 (N/A)
Convolution10	5.10	5.10	None
dw_1x1_fire6_fire6	8.7	1.7	7840 (N/A)
Convolution12	5.10	5.10	None
dw_1x1_fire7_fire7	8.7	1.7	2548 (N/A)
Convolution14	5.10	5.10	None
dw_1x1_fire8_fire8	8.7	1.7	980 (N/A)
E_dense/BiasAdd	8.7	5.10	256 (N/A)

Figure 6.6. Q Format Settings for Each Layer

10. Double-click **Compile** to generate the firmware and filter binary file.

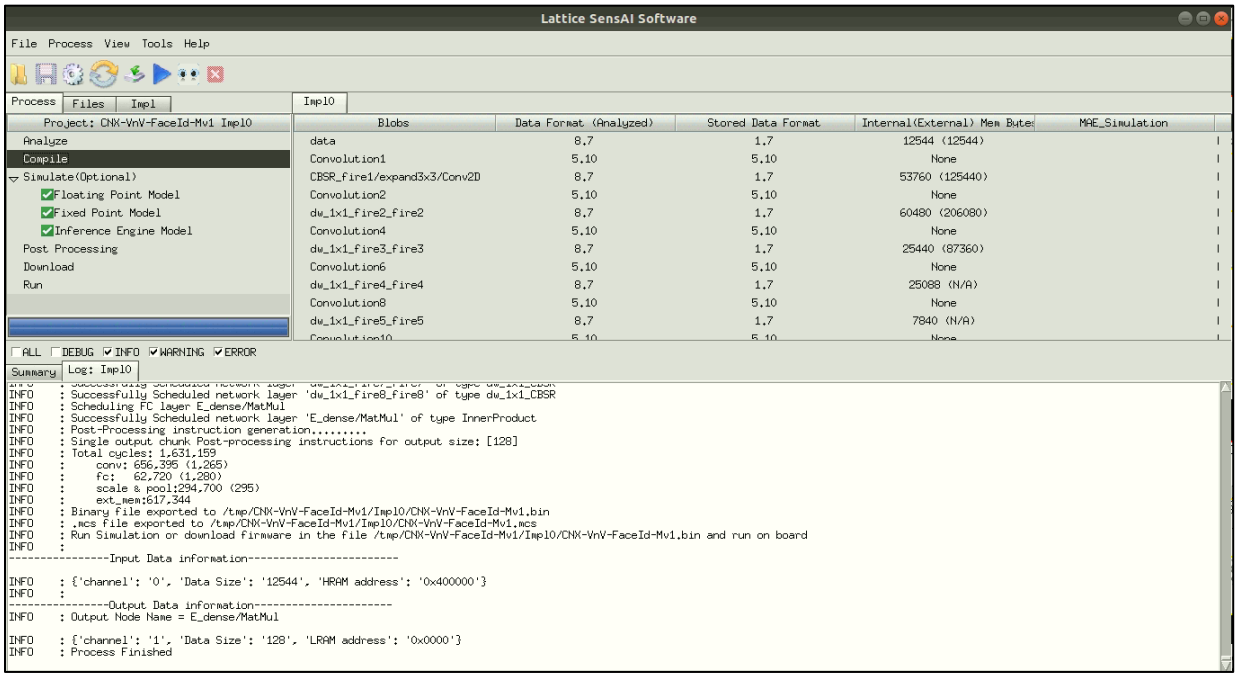

Figure 6.7. Compile Project

The Firmware bin file location is displayed in the compilation log. Use the generated firmware bin on hardware for testing.

7. Hardware Implementation

7.1. Top Level Information

7.1.1. Block Diagram

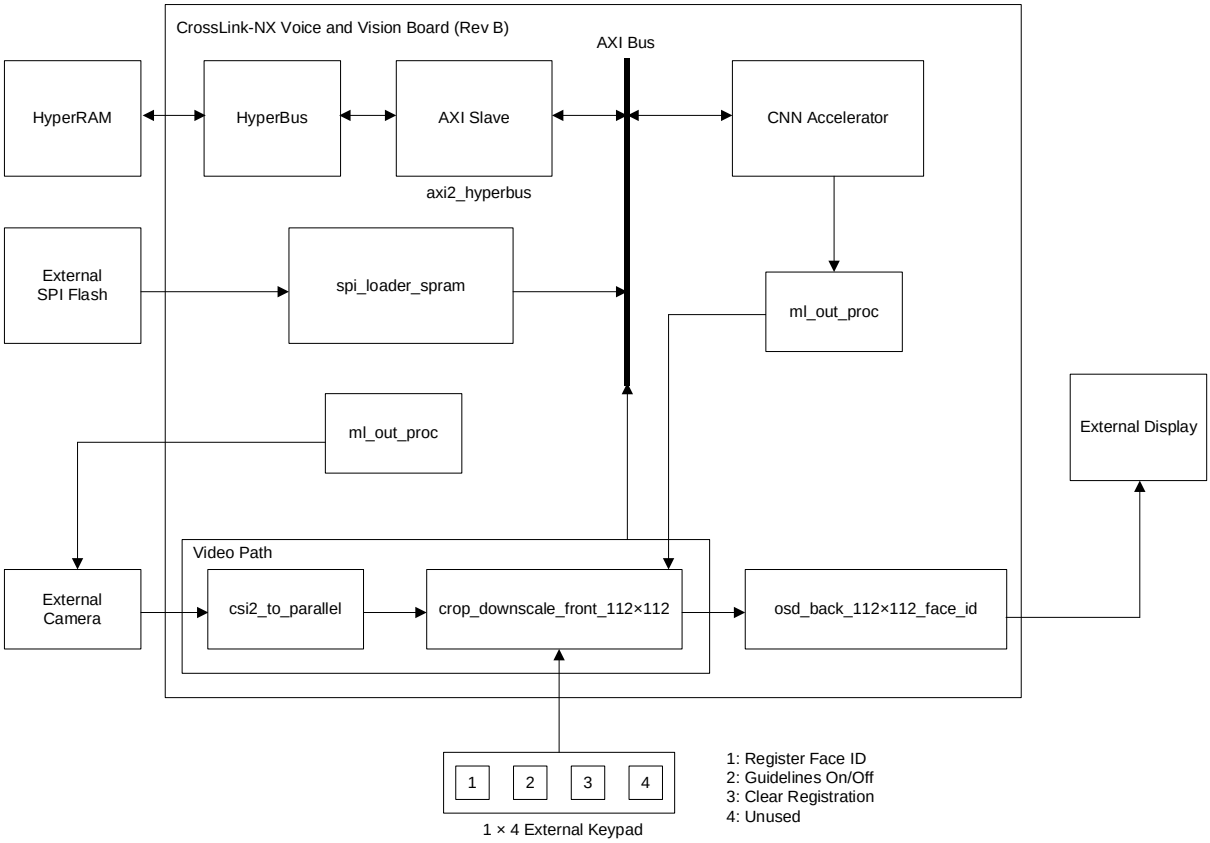

Figure 7.1. RTL Top Level Block Diagram

7.1.2. Push Buttons for Face ID Demo

Figure 7.2 shows the CrossLink-NX VVML board (Rev B). As shown in Figure 7.2, the SW2 push button is used for Registration and SW3 is used for clearing previously registered data. Refer to Push Button Control section for more information.

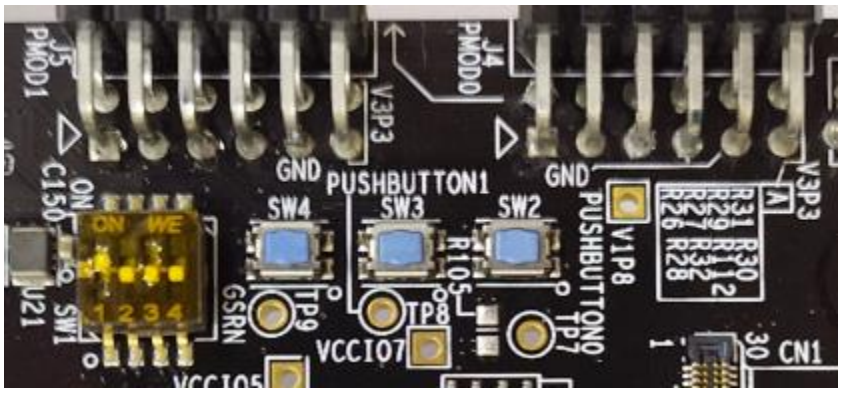

Figure 7.2. Push Buttons on CrossLink-NX VVML Board for Face ID Demo

7.1.3. Operational Flow

- The CNN module is configured with the help of a binary (mcs) sequence command code file, which is generated by the Lattice Machine Learning software tool.
- The command code is written to *spi_loader_spram*, which further stores it in HyperRAM through AXI before the execution of CNN Accelerator IP Core starts.
- The external camera configured using i2c_single logic block captures the raw image and shares it to the *csi2_to_parallel* module. This module separates the R, G, and B pixels from the raw data.
- The RGB data from the *csi2_to_parallel* module is downscaled to 112 × 112 image resolution by the *crop_downscale_front_112x112* module to match CNN's input resolution. This data is written into HyperRAM memory through *axi2_hyperbus* through the *axi_ws2m* AXI interface module.
- After the command code and input data are available to CNN Accelerator from HyperRAM, the IP Core starts inference at the rising edge of ML start signal.
- The push button control logic is managed by the *crop_downscale_front_112x112* module. The registered Face IDs are stored by the *ml_out_proc* module.
- The output data of CNN is passed to *ml_out_proc* for post processing. ml_out_proc now compares the CNN results with initially stored face identifications and passes valid detected Face ID/index and distance to external display module *osd_back_112x112_face_id* through the *crop downscale* module. The output display can be observed in the video player AMCap software.

7.1.4. Core Customization

Table 7.1. Core Parameter

Constant	Default (Decimal)	Description
NUM_FEAT	128	Number of features available in CNN output
FRAC_POS	10	Fraction Part Width in Q-Format representation
DIST_THRESH	45	Indicates Distance Threshold value for Face ID detection output
EN_INF_TIME	0	Enable Timing measurement logic By default, it is zero and the memory file used is *face_id_dissplay.mem*. If assigned 1, timing measurement is enabled and the memory file used is *face_id_display_INF.mem*. In order to configure the respective memory file, follow the below steps 1. Open *dpram8192x8.ipx* from the File List in Radiant. 2. Click *Browse Memory File* from the Initialization section. 3. Update mem file path: • For 0 – /src/jedi_common/face_id_display.mem • For 1 – /src/jedi_common/face_id_display_INF.mem
INF_MULT_FAC	19883	Inference time multiplying factor calculated as per CNN clock frequency and using Q-Format (Q1.31). CNN Clock Frequency = 108 MHz Hence, CNN clock period = $1/(108 \times 10^{-6})$ µs = 0.000009259 ms Now, Q1.31 = $0.000009259 \times 2^{31}$ = ~19883
FLASH_START_ADDR	24'h300000	SPI Flash Read Start Address (keep the same address in Programmer while loading the firmware file) For example, for the current start address, programmer address should be: 0x00300000.
FLASH_END_ADDR	24'h700000	SPI Flash Read End Address (keep the same address in Programmer while loading the firmware file). The address must be in multiple of 512 bytes. For example, for the current end address, programmer address should be: 0x00700000.

Constant	Default (Decimal)	Description
Constant Parameters (Not to be modified)		
PIC_WIDTH	112	Picture Pixel Width (CNN Input)
PIC_HEIGHT	112	Picture Pixel Height (CNN Input)
HYPERRAM_BASEADDR	32'h400000	Indicates HyperRAM starting Base address location value. This should match in the sensAI compiler while generating the firmware.

7.2. Architecture Details

7.2.1. SPI Flash Operation

The RTL module *spi_loader_spram* provides SPI Flash read operation and writes that data into HyperRAM through the AXI interface. It reads from SPI Flash as soon as the board gets powered up, the .bit and .bin files are loaded in the expected addresses.

- Expected Address for .bit file (Programmer) – 0x0000000 - 0x00110000
- Expected Address for Firmware file (Programmer) – FLASH_START_ADDR to FLASH_END_ADDR

Typical sequence of SPI Read commands for SPI Flash MX25L12833F is implemented using FSM in RTL as per below flow of operation.

- After FPGA Reset, RELEASE FROM DEEP POWER DOWN command (0xAB) is passed to SPI Flash memory. RTL then waits for 500 clock cycle for SPI flash to come into Stand By mode, if it is in Deep Power Down mode.
- RTL sends FAST READ command code (0x0B) on SPI MOSI signal for indication of Read Operation to SPI Flash.
- RTL sends three Bytes of Address on SPI MOSI channel which determines the location in SPI flash from the position the data needs to be read.
- This SPI Flash has eight Dummy cycles as wait duration before read data appears on MISO channel. After waiting for eight dummy cycles, the RTL code starts reading data.
- This read sequence is shown in Figure 7.3. The SPI Interface Signal Mapping with RTL signals are as follows
  - CS (Chip Select) => SPI_CSS
  - SCLK (Clock) => SPI_CLK
  - SI (Slave In) => SPI_MOSI
  - SI (Slave Out) => SPI_MISO
- The Read Data on MISO signal is stored in a FIFO in RTL, which then reads the data in multiple of 512 bytes. After 512 bytes chip select is deasserted, the AXI FSM state is activated.

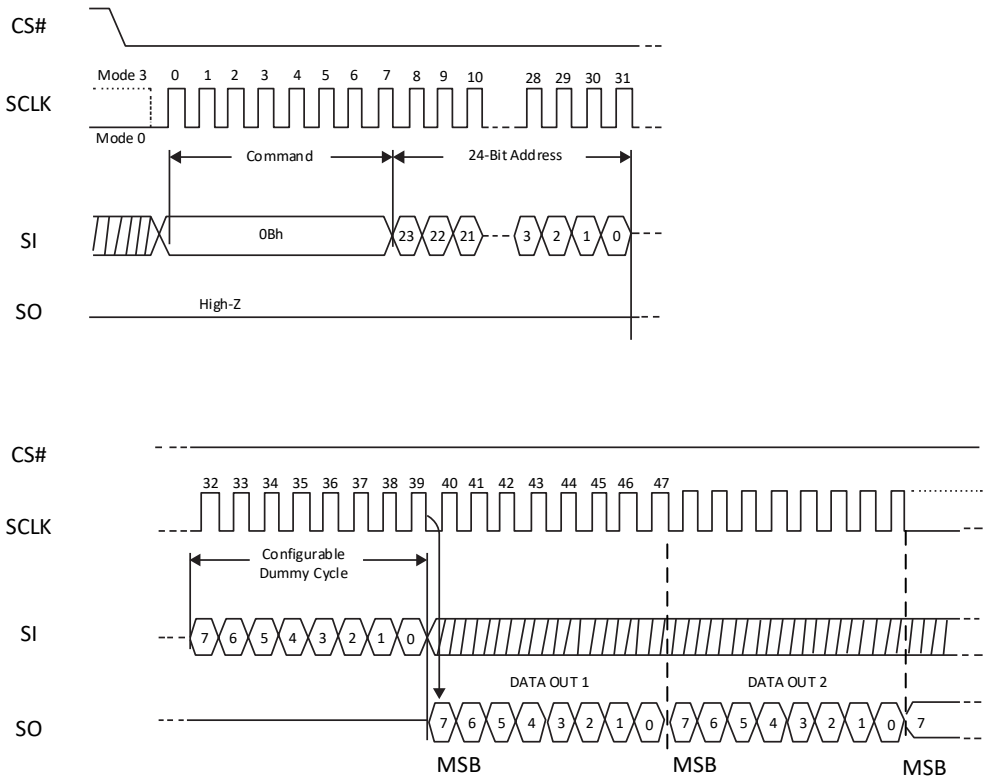

Figure 7.3. SPI Read Command Sequence

- AXI logic reads the data from FIFO in bursts of four on the AXI write channel, with each burst having 128 bytes.
- In accessing the HyperRAM, the *axi_ws2m* module is used as a Muxing module among the multiple input slave AXI interfaces as shown in Figure 7.6. The *spi_loader_spram* module is considered as SLAVE 0 and given priority to write into HyperRAM. The MASTER Interface connects to the *axi2_hyperbus* module, which provides output interface for accessing HyperRAM.
- After writing to HyperRAM is complete, the 512 bytes are fetched from the SPI Flash using the same command sequence as explained above until the FLASH_END_ADDR is reached.

7.2.2. Pre-processing CNN

The pre-processing operations are mainly handled by the *crop_downscale_front_112x112* module as mentioned below:

- This module mainly crops the incoming image to 224 × 224 image data and further generates downscaled input of 112 × 112 image data for the CNN IP input.
- Manages the control logic for SW2 and SW3 push buttons.
- Provides content for the output screen display to the *osc_back_112x112_face_id* module
- Calculates the inference time of theCNN IP

The output screen display and inference time management are explained in the Output Screen Display section**.**

7.2.2.1. Masking and Downscaling flow

- Image data values for each pixel are fed serially line by line for an image frame from *csi2_to_parallel* block.
- These values are considered as valid only when horizontal and vertical masks are inactive. Mask parameters set to mask out boundary area of VGA resolution 640x480 to 224x224 are shown below.
  - Left masking start = 208
  - Right masking end = 432 (Obtained as 208+224)
  - Top masking start = 128
  - Bottom masking end = 352 (Obtained as 128+224)

- Figure 7.4 shows the image obtained after masking.

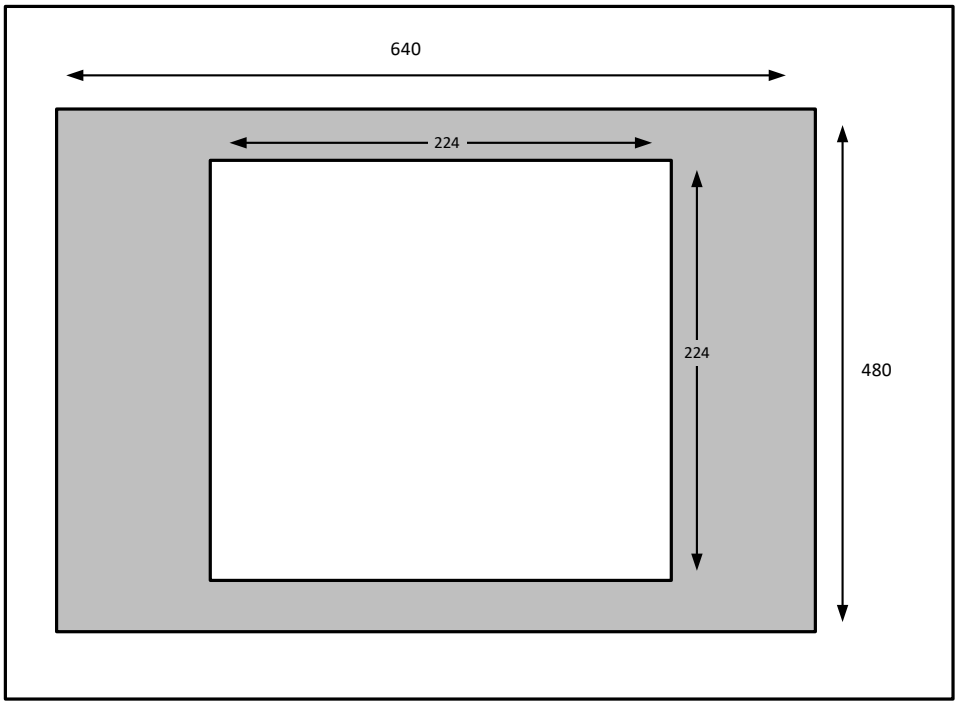

Figure 7.4. Masking

- This 224 × 224 frame image is downscaled into a 112 × 112 resolution image as shown in the **Figure 7.5** by accumulating 2 × 2 pixels into single pixel. For example, 224/2 × 224/2 = 112 × 112.

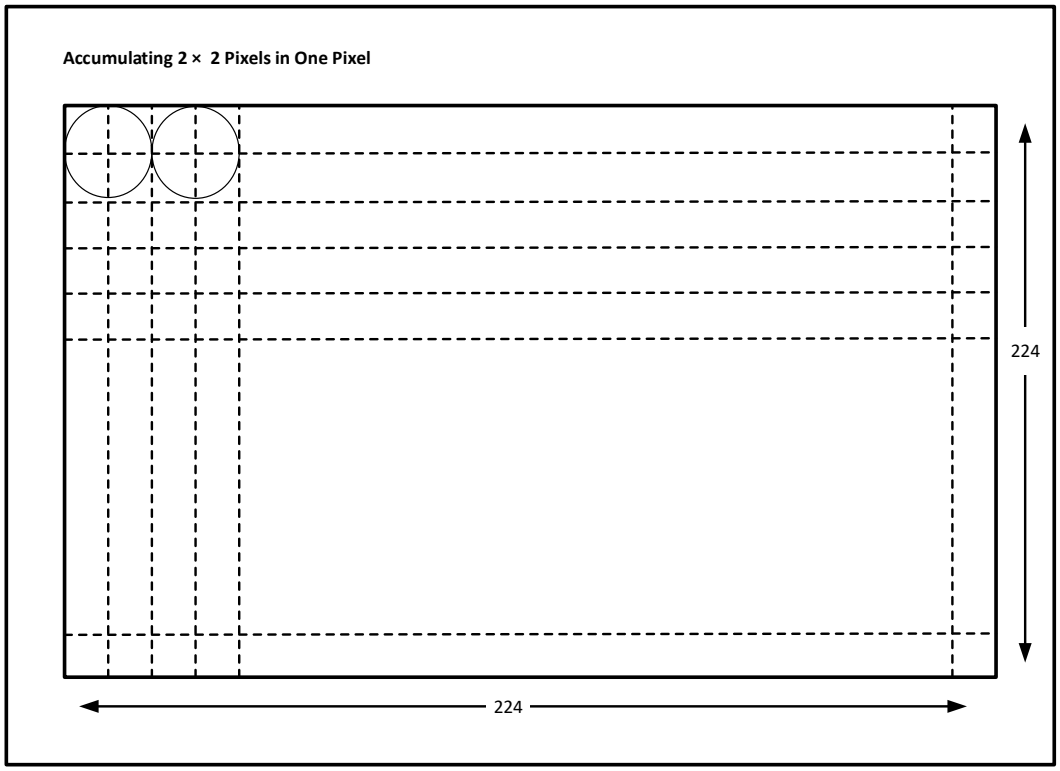

Figure 7.5. Downscaling

- The accumulated value is written in the Frame Buffer, which is a True Dual-Port RAM. The accumulated pixel values for the 2 × 2 grids are stored in the same memory location. When data is read from memory, each RGB value is divided by four (that is, the area of the 2 × 2 grid) to take the average of the 2 × 2 grid matrix.

The data from memory is read and stored in HyperRAM for CNN input through *axi2_hyperbus*, through the *axi_w2sm* module which acts as an AXI interface to write data from slave (*crop_downscale_front_112x112*) to the master HyperRAM. This process is described in the HyperRAM Operations section.

7.2.2.2. Push Button Control

The information of the SW2 and SW3 push buttons connected on CrossLink-NX VVML board is passed into this module. Based on the button pressed, this module defines the following operational modes:

Table 7.2. Push Button Modes

Push Buttons	Mode	Description
SW2	REG	Indicates Registration mode.
SW3	CLEAR	Indicates Clear mode.

- The registration mode is used to register the user faces for identification purposes. A maximum eight user faces can be registered.
- The check mode is used by the demo to check/identity the new face by comparing with the previously registered face, by default, when no button is pressed.
- The green guide lines are always enabled in display. They mainly indicate the active area inside the 224 × 224 frame, where in you have to keep the face close to the camera for registration.
- The clear mode is used to clear all registrations and previous output results. Upon pressing the SW3 push button, the values displayed in Output also gets cleared and you can now initiate new registration process again.

This module manages the logic to Calculate Inference time of CNN IP as per parameter EN_INF_TIME passed from design top. It also collects the outputs from the post-processing module, inference time values, and shares the same with the output screen display block for display in the AMCap video software.

7.2.3. HyperRAM Operations

The CrossLink-NX VVML board uses external HyperRAM for faster data transfer mechanism among the internal blocks and enhances the system performance. The *crop_downscale_front_112x112* module uses HyperRAM to store the downscaled image data.

- The 640 × 480 image is distributed into 224 horizontal and 224 vertical lines, and each block consists of 2 × 2 pixels as shown in Figure 7.5. Thus, there are total 112 × 112 pixel values for the downscaled image.
- Primarily, the *crop_downscale_front_112x112* module stores 112 values into a local FIFO for all 112 horizontal blocks. Later, this stored data is written to HyperRAM through the AXI write data channel.
- When final data is written out, 112 × 112 pixels are initially stored into HyperRAM starting from the base address 0x400000.
- The 112 × 112 pixel values stored in HyperRAM are serially obtained by the CNN engine after getting command sequence through the AXI interface.
- In order for the *crop_downscale_front_112x112* module to access HyperRAM for operations explained above, the *axi_ws2m* module functions as a Muxing module for multiple input slave AXI interfaces as shown in Figure 7.6.
- For the internal blocks to access HyperRAM, the *axi_ws2m* module considers the *spi_loader_spram* module as SLAVE 0, the *cnn_opt* module as SLAVE 1, *crop_downscale_front_112x112* module as SLAVE2, and the MASTER connects these slaves to the *axi2_hyperbus* module.
- The priority to select write channel is given, respectively, to the SPI loader slave, CNN slave, and crop-downscale slave. Whenever valid address is available from the respective Slave on its write address channel, that slave is given access of master channel if other priority slaves are not accessing it. Thus, when valid write address is obtained from the *crop_downscale_front_112x112* module, access is given to Slave 2 to use HyperRAM.

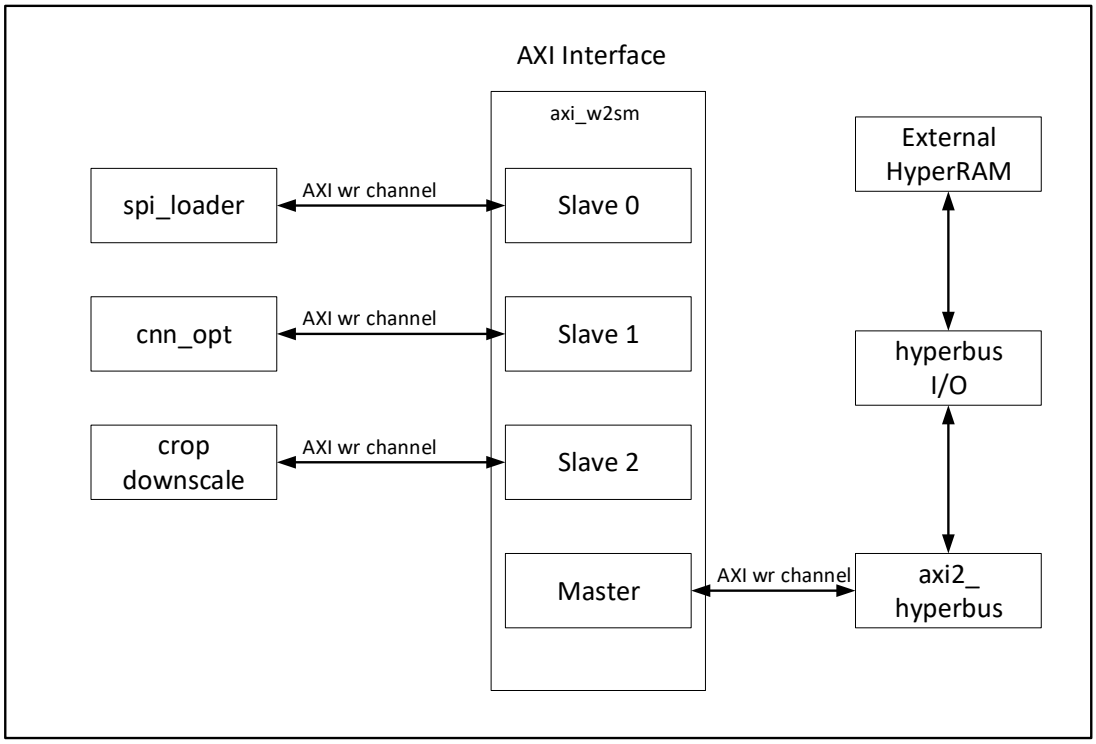

Figure 7.6. HyperRAM Access Block Diagram

7.2.4. Post-Processing CNN

The primary functionality of this block (*ml_out_proc*) is to capture the CNN valid output, detect, or register face identification information and pass valid face index and distance back to the *crop_downscale_front_112x112* module for display.

Table 7.3. Post-Processing Parameters

Constant	Default (Decimal)	Description
NUM_FEAT	128	Number of Features (values) provided by CNN for each processed frame
FRAC_POS	10	Fraction Part Width as per Q-Format 5.10 representation of last output layer
DIST_THRESH	45	Upper Threshold for Mean Squared Difference (distance) value calculated for Face Features

CNN provides 128 values (16-bit each) representing the Face Features for each frame processed. These values are processed according to the mode selected (REG/CHECK/CLEAR), as explained in further sections.

7.2.4.1. REG MODE

- This mode is used to register/store the values of face measurements provided by CNN into memory.
- The stored feature values are used as a reference while CHECK Mode is on.
- This module can store up to eight face identification measurements. Each Face Identification measurement uses separate 8k RAM.

7.2.4.2. CHECK MODE

- This mode is used to detect the valid face identification matching with current face from the stored face identifications in REG MODE.
- It compares the stored face identification (0-7) feature values with the current CNN output feature values and performs mean square difference to estimate the average distance from actual values for each feature.

- Final accumulated mean square difference value for each Face Identification is compared with each other and the index with least difference value is selected.
- If the difference value is less than DIST_THRESH, the selected face index is valid and is having minimum distance from current face.
- The matched face index and all distance values are passed to crop downscale module for display. If no face identification matches (all calculated distance values > DIST_THRESH OR no face identification is registered by REG MODE), then the output face index is passed with invalid value (0xF), displayed as *"?"*.

7.2.4.3. CLEAR MODE

When this mode is activated, all registered face index and distance information are cleared. You can now register new faces using the REG mode.

7.2.5. Output Screen Display

- *osd_back_112x112_face_id* is responsible to manage output display. It has the *lsc_osd_text* module which provides the bitmap (from rom_8x8font) of each ASCII character to be displayed with specified position on screen. For the text display of result values, it receives inputs as number of registered Face IDs, valid Face ID detected, distance of each entry, and inference time values (as per enabled/disabled) from the crop downscale module. The texts obtained on the left side in display are pre-stored in the form of their respective ASCII Character values in the (*face_id_display.mem/face_id_display_INF.mem*) files which are loaded in dpram8192x8 before exporting the bit files.
- This sets an output signal (*text_on*) when text is displayed. When *text_on* is set, the value for that pixel location is assigned FFF value (white color) and sent to the AMCap video display software through the FX3 USB Wrapper module.

7.2.5.1. USB Wrapper

- The *Wrapper_USB3* module is used for to transmit 16-bit data to the output 16-bit interface every clock cycle.
- This module takes the input data in YCbCr 24-bit format and gives output as 16-bit YCb and YCr format. This module does not change or regenerate input timing parameters.

7.2.5.2. Inference Time Calculation

The time taken by a trained neural network model to infer/predict outputs after obtaining input data is called inference time. The process of this calculation is managed by the crop downscale module explained as follows.

- The inference time is calculated by implementing a counter to store the count of CNN engine cycles per frame.
- When the *i_rd_rdy* signal (that is *o_rd_rdy* coming from CNN engine) is high, the CNN engine indicates that it is ready to get input; and when it is low, the engine indicates that it is busy.
- When the *i_rd_rdy* signal is low, the CNN counter begins and stops when the *i_rd_rdy* signal goes high again indicating that previous execution is over and the CNN is ready for new input.
- As shown in Figure 7.7, when *rdy_h2l* (ready high-to-low) pulse is asserted, the CNN Up-counter starts from 1 and the count value increases until *i_rd_rdy* is not high again. The count value is stored in *count*.
- Similarly, when *rdy_l2h* (ready low-to-high) pulse is asserted, the Up-counter stops and the final CNN count value is obtained *cnn_count*.

```
assign rdy_l2h          =  i_rd_rdy & ~rdy;
assign rdy_h2l          = ~i_rd_rdy & rdy;

// Counter to calculate individual frame inference time
assign count_c          = (o_rd_done & rdy_h2l) ? 27'd1 : (i_rd_rdy) ? count : (count + 27'd1);

// Store individual cnn frame counter
assign cnn_count_c      =  rdy_l2h ? count : cnn_count;
```

Figure 7.7. CNN Counter Design

The methodology used to obtain stable inference time is to calculate inference time per frame and obtain the average inference time value after 16 CNN frames are over as discussed below.

- After completion of every frame, the new count value *cnn_count* obtained, as explained above, is added to the previous value and stored in *cnn_adder*.
- A frame counter monitors the frame count. After 16 frames, when the frame count is done, the *cnn_adder* value is reset as shown in Figure 7.8.

```
// Frame counter to calculate CNN frames upto 16 (0 - 15)
assign frame_counter_c   = (rdy_l2h_rr)? (frame_counter + 4'd1) : frame_counter;

// keep adding indiviual cnn frame counter for 16 frames. Then clear to 0
assign cnn_adder_c       = (count_done)? 31'd0 : (rdy_l2h_r) ? (cnn_adder + {4'd0,cnn_count}) : cnn_adder;

// Counter addition done when 1ll 16 cnn frame counter values are added and averaged
assign count_done        = (frame_counter == 4'd15) & (rdy_l2h_rr);
```

Figure 7.8. Frame Counter Design for 16 CNN Frames Average

- To get the average inference time value *avg_inf_time_hex* after frame count is done, the final *cnn_adder* value is divided by 16 as shown in Figure 7.9.

```
// Average Inference Time calculated by dividing by 16
assign inf_time_c        = (count_done)? cnn_adder[30:4] : inf_time;

// 32 Bit average Inference time in Hex
assign avg_inf_time_hex  = {5'd0,inf_time};
```

Figure 7.9. Average Inference Time Calculation

- Using the Lattice Multiplier library module, the average inference time value is multiplied by *INF_MULT_FAC*. A parameter indicating inference multiplying factor explained in Table 7.1.
- The inference time in millisecond *inf_time_ms* is obtained by dividing the output obtained from this multiplier by 2^31 as per the Q-Format, shown in Figure 7.10.
- All the above obtained values namely the CNN count, the average inference time, and the inference time in millisecond are passed on to the *lsc_osd_text* module for getting bitmap to the display characters.

```
assign inf_time_ms       = inf_time_mult[46:31];
```

Figure 7.10. Inference Time in Millisecond

7.2.5.3. Inference Time Display Management

The Inference Time Display Management module mainly consists of a DPRAM, which holds the characters at pre-defined address positions indicated by *text_addr* and an 8 × 8 font ROM which provides the bitmap of these characters for PC display.

This module basically functions by using two entities. One is the position of the character where it has to be displayed, and the other is by reading the ASCII value of the character to be displayed.

For this purpose, once the CNN count, individual frame inference time and the inference time in millisecond values are obtained, they are converted from hex into ASCII values as shown in Table 7.4.

The average inference time input value *i_avg_inf_time_hex* is converted from hex to ASCII values as shown below. To display eight characters of this value on PC, this input is stored in the respective *r_avginfhex_ch*. The characters obtained by adding 7'h30 and 7'h37 are shown in Figure 7.11.

```
r_avginfhex_ch0 <= (i_avg_inf_time_hex[31:28] > 4'd9)? (i_avg_inf_time_hex[31:28] + 7'h37) : (i_avg_inf_time_hex[31:28] + 7'h30);
r_avginfhex_ch1 <= (i_avg_inf_time_hex[27:24] > 4'd9)? (i_avg_inf_time_hex[27:24] + 7'h37) : (i_avg_inf_time_hex[27:24] + 7'h30);
r_avginfhex_ch2 <= (i_avg_inf_time_hex[23:20] > 4'd9)? (i_avg_inf_time_hex[23:20] + 7'h37) : (i_avg_inf_time_hex[23:20] + 7'h30);
r_avginfhex_ch3 <= (i_avg_inf_time_hex[19:16] > 4'd9)? (i_avg_inf_time_hex[19:16] + 7'h37) : (i_avg_inf_time_hex[19:16] + 7'h30);
r_avginfhex_ch4 <= (i_avg_inf_time_hex[15:12] > 4'd9)? (i_avg_inf_time_hex[15:12] + 7'h37) : (i_avg_inf_time_hex[15:12] + 7'h30);
r_avginfhex_ch5 <= (i_avg_inf_time_hex[11:8]  > 4'd9)? (i_avg_inf_time_hex[11:8]  + 7'h37) : (i_avg_inf_time_hex[11:8]  + 7'h30);
r_avginfhex_ch6 <= (i_avg_inf_time_hex[7:4]   > 4'd9)? (i_avg_inf_time_hex[7:4]   + 7'h37) : (i_avg_inf_time_hex[7:4]   + 7'h30);
r_avginfhex_ch7 <= (i_avg_inf_time_hex[3:0]   > 4'd9)? (i_avg_inf_time_hex[3:0]   + 7'h37) : (i_avg_inf_time_hex[3:0]   + 7'h30);
```

Figure 7.11. Average Inference Time Value to ASCII Conversion

Table 7.4. Signal Values to ASCII Conversion

Characters for Display	Value to be Added To Signal	ASCII Hex Value	ASCII Decimal Value
1	7'h30	31	49
2	7'h30	32	50
3	7'h30	33	51
4	7'h30	34	52
5	7'h30	35	53
6	7'h30	36	54
7	7'h30	37	55
A	7'h37	41	65
B	7'h37	42	66
C	7'h37	43	67
D	7'h37	44	68
E	7'h37	45	69
F	7'h37	46	70

To display eight characters of individual frame inference time, the input signal *i_inf_time_hex* is converted from hex to ASCII and stored in respective *r_infhex_ch* signal as shown in Figure 7.12.

In the same way, to display four characters of inference time in ms, the input signal *i_inf_ms* is converted from hex to ASCII and stored in the respective *r_inf_ms* signal as shown in Figure 7.13.

```
r_infhex_ch0    <= (i_inf_time_hex[31:28] > 4'd9)? (i_inf_time_hex[31:28] + 7'h37) : (i_inf_time_hex[31:28] + 7'h30);
r_infhex_ch1    <= (i_inf_time_hex[27:24] > 4'd9)? (i_inf_time_hex[27:24] + 7'h37) : (i_inf_time_hex[27:24] + 7'h30);
r_infhex_ch2    <= (i_inf_time_hex[23:20] > 4'd9)? (i_inf_time_hex[23:20] + 7'h37) : (i_inf_time_hex[23:20] + 7'h30);
r_infhex_ch3    <= (i_inf_time_hex[19:16] > 4'd9)? (i_inf_time_hex[19:16] + 7'h37) : (i_inf_time_hex[19:16] + 7'h30);
r_infhex_ch4    <= (i_inf_time_hex[15:12] > 4'd9)? (i_inf_time_hex[15:12] + 7'h37) : (i_inf_time_hex[15:12] + 7'h30);
r_infhex_ch5    <= (i_inf_time_hex[11:8]  > 4'd9)? (i_inf_time_hex[11:8]  + 7'h37) : (i_inf_time_hex[11:8]  + 7'h30);
r_infhex_ch6    <= (i_inf_time_hex[7:4]   > 4'd9)? (i_inf_time_hex[7:4]   + 7'h37) : (i_inf_time_hex[7:4]   + 7'h30);
r_infhex_ch7    <= (i_inf_time_hex[3:0]   > 4'd9)? (i_inf_time_hex[3:0]   + 7'h37) : (i_inf_time_hex[3:0]   + 7'h30);
```

Figure 7.12. CNN Count Values to ASCII Conversion

```
r_infms_ch0     <= (i_inf_time_ms[15:12] > 4'd9)? (i_inf_time_ms[15:12] + 7'h37) : (i_inf_time_ms[15:12] + 7'h30);
r_infms_ch1     <= (i_inf_time_ms[11:8] > 4'd9)? (i_inf_time_ms[11:8] + 7'h37) : (i_inf_time_ms[11:8]  + 7'h30);
r_infms_ch2     <= (i_inf_time_ms[7:4] > 4'd9)? (i_inf_time_ms[7:4] + 7'h37) : (i_inf_time_ms[7:4] + 7'h30);
r_infms_ch3     <= (i_inf_time_ms[3:0] > 4'd9)? (i_inf_time_ms[3:0] + 7'h37) : (i_inf_time_ms[3:0] + 7'h30);
```

Figure 7.13. Inference Time in Millisecond Values to ASCII Conversion

The positions where these values have to be displayed are given using the *text_addr* signal as shown in Figure 7.14. The use of these locations is shown in Figure 7.14. A memory initialization file *face_id_display_INF.mem* is used by the Lattice Radiant tool to store characters at address locations for display.

```
// Inference Time Logic
// Hex counter display
assign w_avginfhex_ch0_pos    = (text_addr == 13'd1135);
assign w_avginfhex_ch1_pos    = (text_addr == 13'd1136);
assign w_avginfhex_ch2_pos    = (text_addr == 13'd1137);
assign w_avginfhex_ch3_pos    = (text_addr == 13'd1138);
assign w_avginfhex_ch4_pos    = (text_addr == 13'd1139);
assign w_avginfhex_ch5_pos    = (text_addr == 13'd1140);
assign w_avginfhex_ch6_pos    = (text_addr == 13'd1141);
assign w_avginfhex_ch7_pos    = (text_addr == 13'd1142);

assign w_infhex_ch0_pos       = (text_addr == 13'd1172);
assign w_infhex_ch1_pos       = (text_addr == 13'd1173);
assign w_infhex_ch2_pos       = (text_addr == 13'd1174);
assign w_infhex_ch3_pos       = (text_addr == 13'd1175);
assign w_infhex_ch4_pos       = (text_addr == 13'd1176);
assign w_infhex_ch5_pos       = (text_addr == 13'd1177);
assign w_infhex_ch6_pos       = (text_addr == 13'd1178);
assign w_infhex_ch7_pos       = (text_addr == 13'd1179);
// Milisecond display
assign w_infms_ch0_pos        = (text_addr == 13'd1145);
assign w_infms_ch1_pos        = (text_addr == 13'd1146);
assign w_infms_ch2_pos        = (text_addr == 13'd1147);
assign w_infms_ch3_pos        = (text_addr == 13'd1148);
```

Figure 7.14. Text Address Positions to Display Input Values

The address location structure for displaying average inference time (of 16 CNN frames) and inference time in millisecond values along with their strings are stored in face_id_display_INF.mem is shown in Figure 7.15.

	Characters stored as ASCII values in face_id_display_INF.mem for String display											Avg Inf Time	face_id_display_INF.mem ASCII values				
Character	A	v	g	l	n	f	T	i	m	e	:		(	*	m	s	)
Address in Decimal	1122	1123	1124	1126	1127	1128	1130	1131	1132	1133	1134	1135 - 1142	1144	1145-1148	1150	1151	1152
(Note: * Indicates the value of Inference Time calculated and displayed in Millisecond)																	

Figure 7.15. Address Locations to Display Individual Frame Time and Inference Time with String in PC

To display the input values in address locations shown in Figure 7.15, the ASCII values obtained as shown in Table 7.4 are sent to the 8 × 8 font ROM with the help of the *font_char* signal to obtain the bitmap for display.

Similarly, like the inference time display, other text characters for the Face ID display like SW2 and SW3 push buttons information, entries, and others are all present in both *.mem* files in the form of ASCII characters at the dedicated address locations.

```
font_char  = (r_num_reg_pos_1d)? r_num_reg_char_1d :
                                 (r_face_id_pos_1d    )? r_face_id_char_1d :
                                 (r_dist_pos_2d       )? r_dist_char       :
                                 (r_avginfhex_ch0_pos )? r_avginfhex_ch0   :   // Inference time display
                                 (r_avginfhex_ch1_pos )? r_avginfhex_ch1   :
                                 (r_avginfhex_ch2_pos )? r_avginfhex_ch2   :
                                 (r_avginfhex_ch3_pos )? r_avginfhex_ch3   :
                                 (r_avginfhex_ch4_pos )? r_avginfhex_ch4   :
                                 (r_avginfhex_ch5_pos )? r_avginfhex_ch5   :
                                 (r_avginfhex_ch6_pos )? r_avginfhex_ch6   :
                                 (r_avginfhex_ch7_pos )? r_avginfhex_ch7   :
                                 (r_infhex_ch0_pos    )? r_infhex_ch0      :
                                 (r_infhex_ch1_pos    )? r_infhex_ch1      :
                                 (r_infhex_ch2_pos    )? r_infhex_ch2      :
                                 (r_infhex_ch3_pos    )? r_infhex_ch3      :
                                 (r_infhex_ch4_pos    )? r_infhex_ch4      :
                                 (r_infhex_ch5_pos    )? r_infhex_ch5      :
                                 (r_infhex_ch6_pos    )? r_infhex_ch6      :
                                 (r_infhex_ch7_pos    )? r_infhex_ch7      :
                                 (r_infms_ch0_pos     )? r_infms_ch0       :
                                 (r_infms_ch1_pos     )? r_infms_ch1       :
                                 (r_infms_ch2_pos     )? r_infms_ch2       :
                                 (r_infms_ch3_pos     )? r_infms_ch3       : r_text_data[6:0];
```

Figure 7.16. Bitmap Extraction from Font ROM

8. Creating FPGA Bitstream File

This section describes the steps to compile RTL bitstream using the Lattice Radiant tool.

8.1. Bitstream Generation Using Lattice Radiant Software

To create the FPGA bitstream file:

1. Open the Lattice Radiant software (v3.0.0.24.1). Default screen in shown in Figure 8.1.

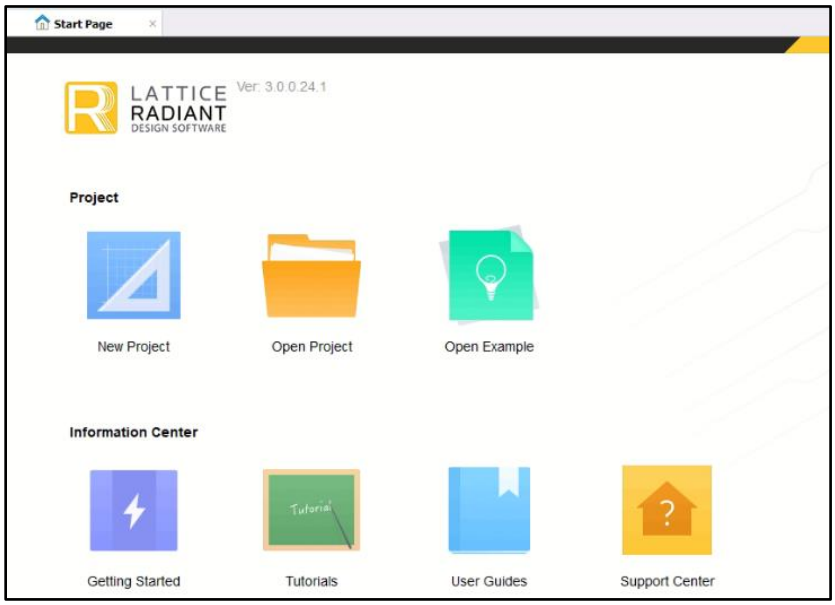

Figure 8.1. Radiant – Default Screen

2. Go to **File > Open > Project**.
3. Open the Radiant project file (.rdf) for CrossLink-NX Voice and Vision Face ID Demo RTL. As shown in Figure 8.2, you can also open project by triggering the yellow folder shown in the user interface.

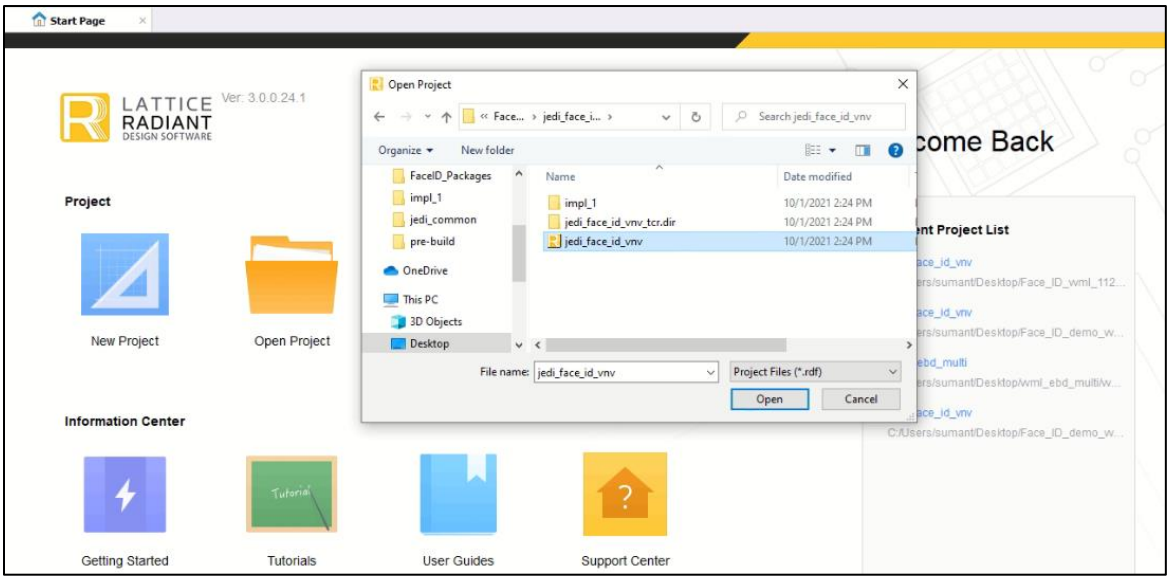

Figure 8.2. Radiant – Open CrossLink-NX Voice and Vision Project File (.rdf)

4. After opening the project file, check the following points in Figure 8.3.
   - Design loaded with zero errors message shown in the Output window below.
   - Check for this information in project summary window.
     - **Part Number – LIFCL-40-7MG289I**
     - **Family – LIFCL**
     - **Device – LIFCL-40**
     - **Package – CSBGA289**

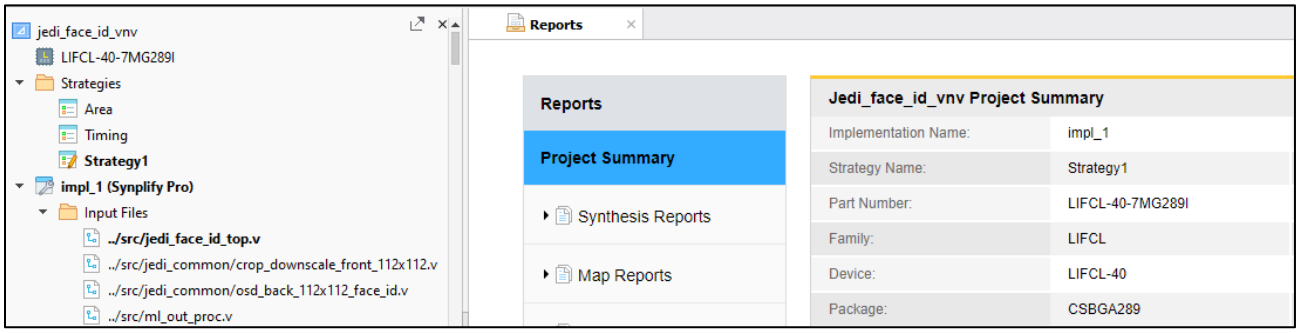

Figure 8.3. Radiant – Design Load Check After Opening the Project File

5. If the design is loaded without errors, click the **Run** button to trigger bitstream generation as shown below in Figure 8.4.

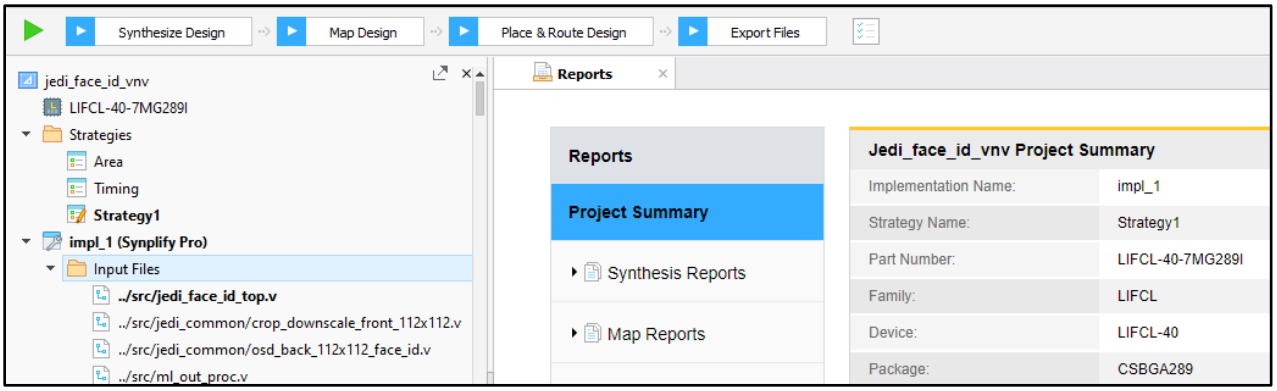

Figure 8.4. Radiant – Trigger Bitstream Generation

6. The Lattice Radiant tool displays *Saving bitstream in …* message in the **Reports** window, as shown in Figure 8.5. The bitstream is generated at *Implementation Location*, as shown in Figure 8.5.

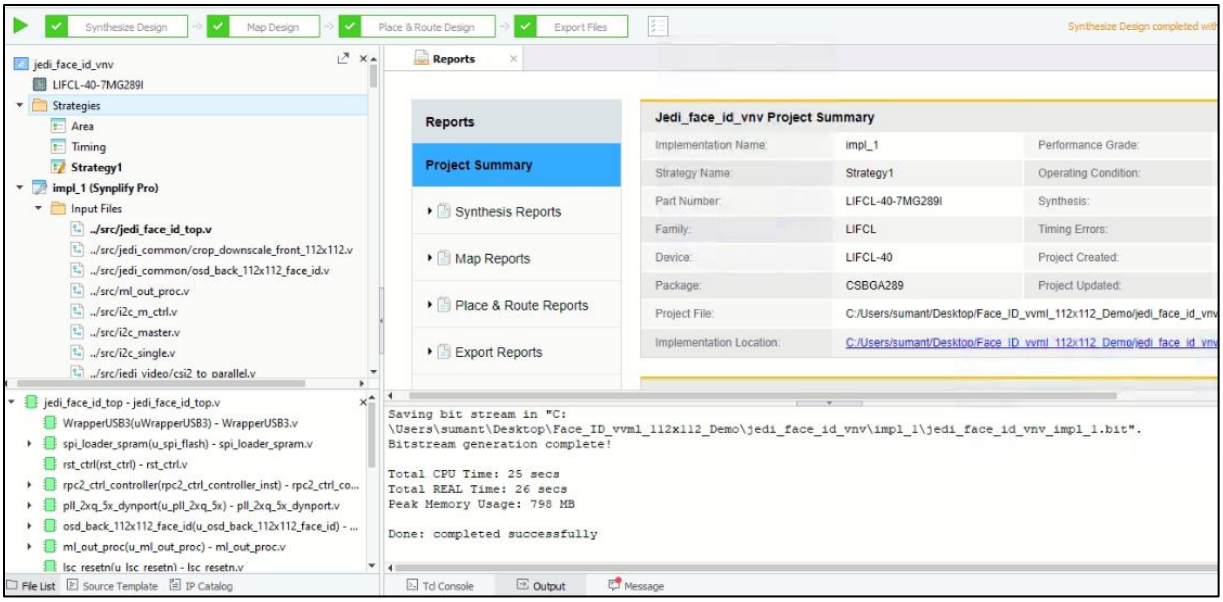

Figure 8.5. Radiant – Bit File Generation Report Window

9. Programming the Demo

This document contains information to run the Face ID demo on the CrossLink-NX-VVML board.

9.1. Package Folder Structure

Figure 9.1 shows the demo folders and files after unzipping the package.

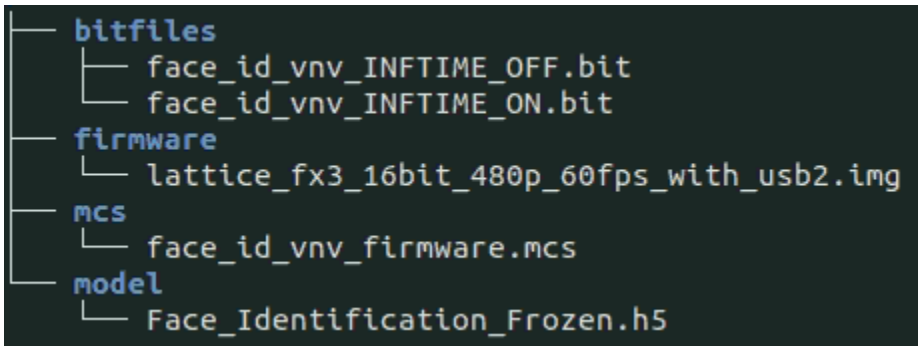

Figure 9.1. Demo Package Folder Structure

9.2. Load Firmware in FX3 I²C EEPROM

To load the firmware:

1. Connect the USB3 port of the CrossLink-NX VVML Board (Rev B) to the PC using the USB3 cable.
2. Connect the Jumper J13 on the board.
3. Open the USB Control Centre application. The Cypress FX3 SDK should be installed.
4. Press the SW2 button to reset the FX3 chip. Figure 9.2 shows the boot loader device screen.

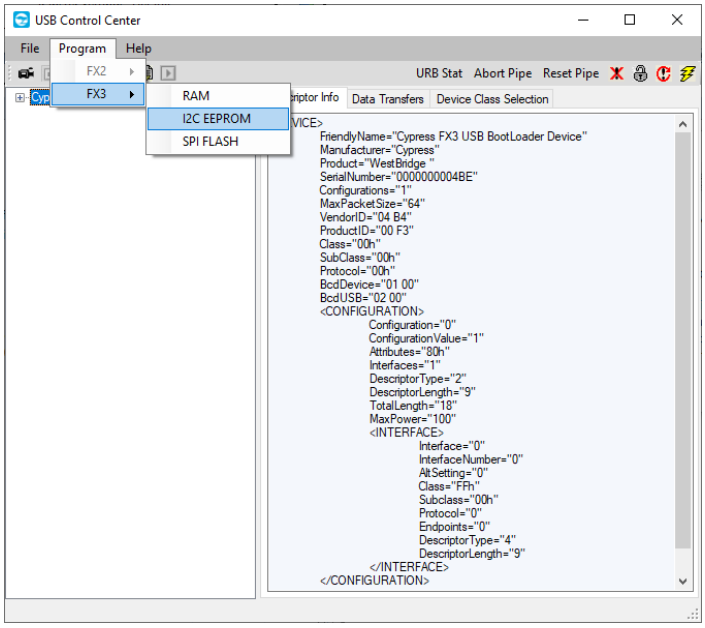

Figure 9.2. Selecting FX3 I²C EEPROM in USB Control Centre

5. Select **Cypress USB Bootloader**.
6. Click **Program > FX3 > I²C E2PROM**.

7. Locate and select the FX3 image file for the 640 × 480p 60 Hz 16 bit configuration.
8. The Firmware is programmed in the I^2C E2PROM.
9. After the operation is completed, a message acknowledging successful programming is shown at the bottom taskbar.
10. Remove jumper J13.
11. Power OFF and then power ON the board.
12. The FX3 boots from I^2C E2PROM.

9.3. Programming the CrossLink-NX Voice and Vision SPI Flash

9.3.1. Erasing the CrossLink-NX Voice and Vision SRAM Prior to Reprogramming

If the CrossLink-NX device is already programmed (either directly, or loaded from SPI Flash), follow this procedure first to erase the CrossLink-NX SRAM memory before re-programming the SPI Flash. If you are doing this, keep the board powered when reprogramming the SPI Flash (so it does not reload on reboot).

To erase the CrossLink-NX device:

1. Launch Lattice Radiant Programmer. In the **Getting Started** dialog box, select **Create a new blank project**.

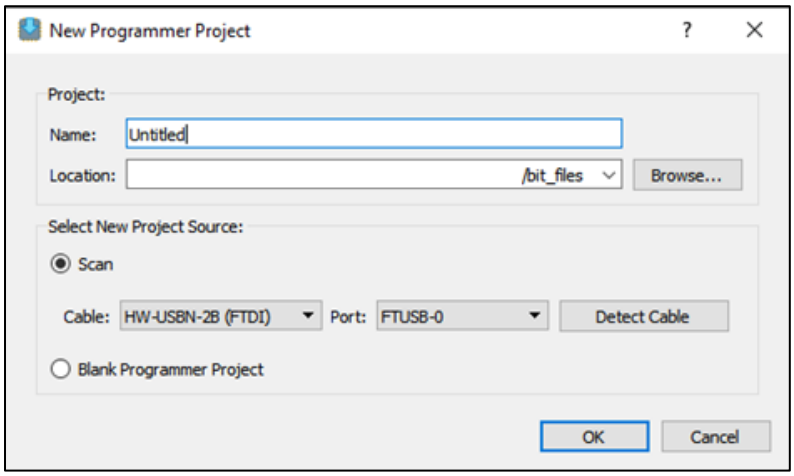

Figure 9.3. Lattice Radiant Programmer Default Screen

2. Click **OK**.
3. In the Lattice Radiant Programmer main interface, select **LIFCL** for Device Family, as shown in Figure 9.4. Select **LIFCL** for Device Vendor and **LIFCL-40** for Device.

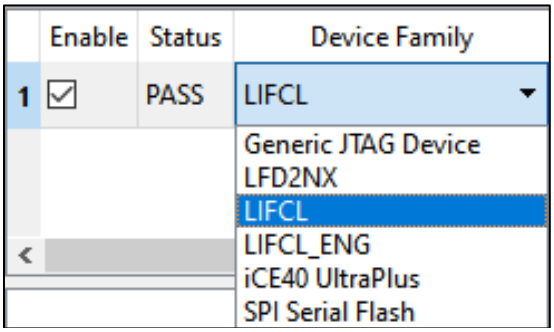

Figure 9.4. Lattice Radiant Programmer- Device Selection

4. Right-click and select **Device Properties**.
5. Select **JTAG** for **Port Interface**, **Direct Programming** for **Access Mode**, and **Erase Only** for **Operation**, as shown in Figure 9.5.

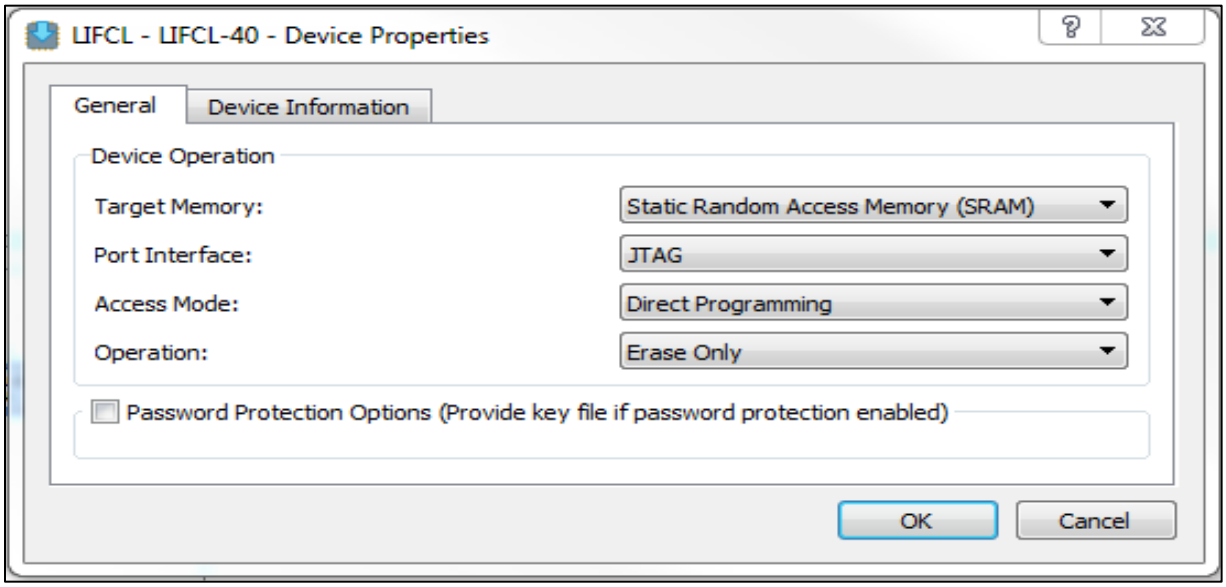

Figure 9.5. Lattice Radiant Programmer – Device Operation

6. Click **OK** to close the **Device Properties** dialog box.
7. Press and hold SW5 until you see the *Successful* message in the Lattice Radiant log window.
8. In the Radiant Programmer main interface, click the **Program** button to start the erase operation.

9.3.2. Programming the CrossLink-NX VVML Board

To program the CrossLink-NX Voice and Vision SPI Flash:

1. Ensure that the CrossLink-NX Voice and Vision device SRAM is erased by performing the steps in the Erasing the CrossLink-NX Voice and Vision SRAM Prior to Reprogramming section.
2. In the Lattice Radiant Programmer main interface, right-click the CrossLink-NX Voice and Vision row and select **Device Properties**.
3. In the **Device Properties** dialog box, apply the settings shown in Figure 9.6.

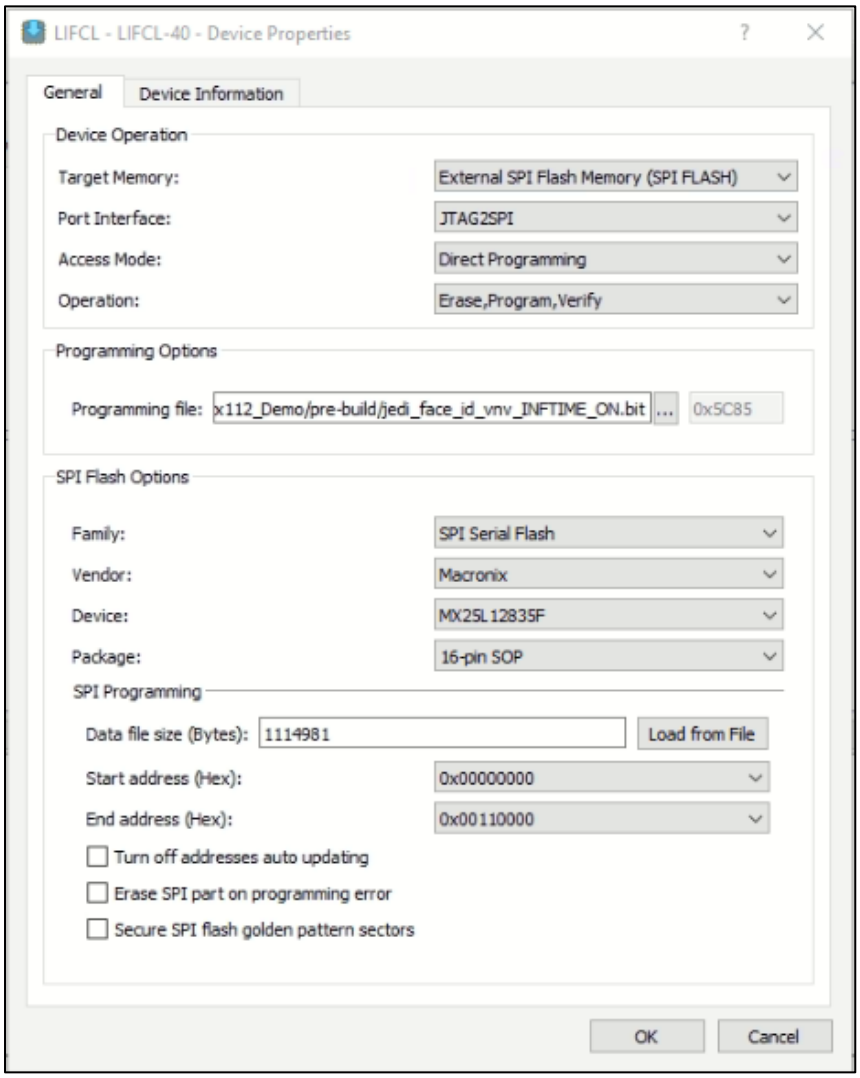

Figure 9.6. Lattice Radiant Programmer – Selecting Device Properties Options for CrossLink-NX Flashing

Notes:

- In **Programming file**, browse and select the CrossLink-NX Voice and Vision bit file (*.bit).
- 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)** – **0x00110000**

4. Click **OK**.
5. Press and hold SW5 until you see the *Successful* message in the Lattice Radiant log window.
6. Click the **Program** button to start the programming operation.
7. After successful programming, the **Output** console displays the result, as shown in Figure 9.7.

Figure 9.7. Lattice Radiant Programmer – Output Console

9.3.3. Programming sensAI Firmware Binary to the CrossLink-NX SPI Flash

To program the CrossLink-NX SPI flash:

1. Ensure that the CrossLink-NX device SRAM is erased by performing the steps in the Erasing the CrossLink-NX Voice and Vision SRAM Prior to Reprogramming section before flashing bitstream and sensAI firmware binary.
2. In the Lattice Radiant Programmer main interface, right-click the CrossLink-NX row and select **Device Properties**.
3. In the **Device Properties** dialog box, apply the settings as shown in Figure 9.8.

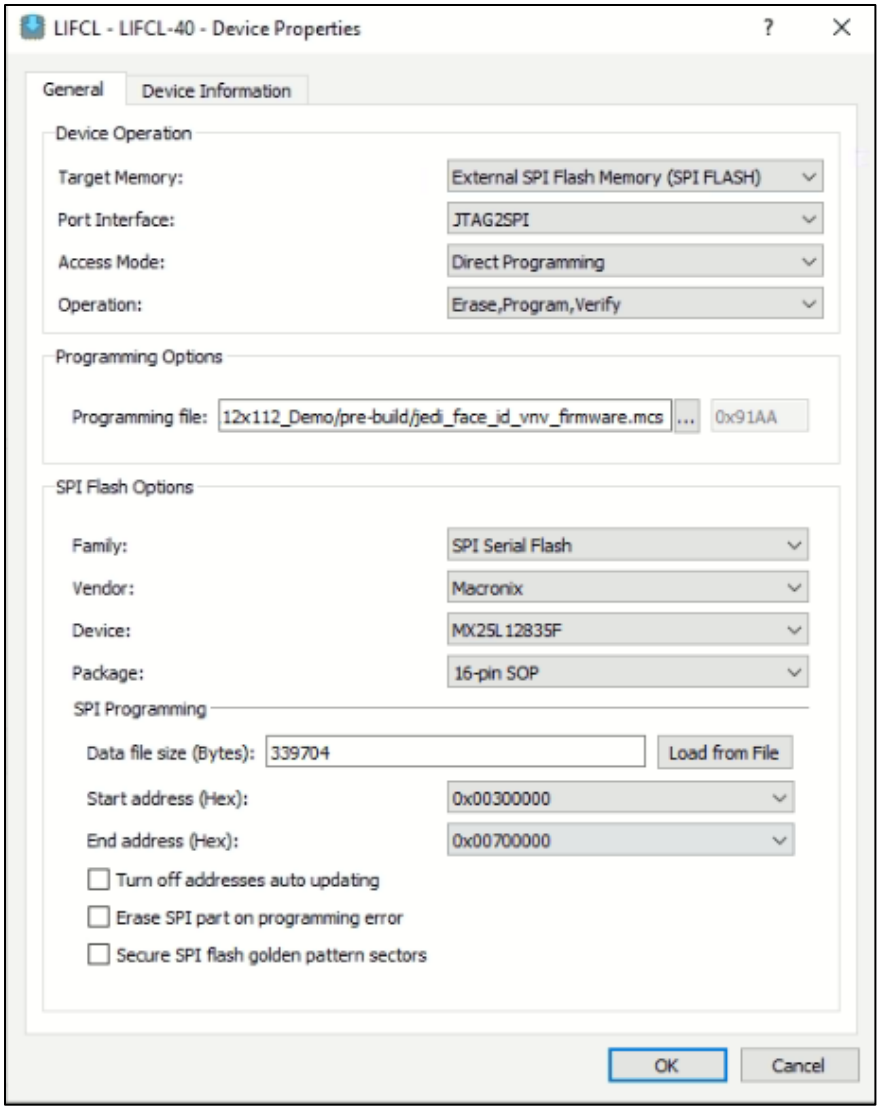

Figure 9.8. Lattice Radiant Programmer – Selecting Device Properties Options for CrossLink-NX Flashing

Notes:

- In **Programming file**, browse and select the CrossLink-NX sensAI firmware binary file after converting it to hex (*.mcs).
- Click **Load from File** to update the Data file size (bytes) value.
- Ensure that the following addresses are correct:
  - **Start Address (Hex) – 0x00700000**
  - **End Address (Hex) – 0x00700000**

4. Click **OK**.
5. Press and hold SW5 until you see the *Successful* message in the Lattice Radiant log window.
6. Click the **Program** button to start the programming operation.
7. After successful programming, the **Output** console displays the result, as shown in Figure 9.9.

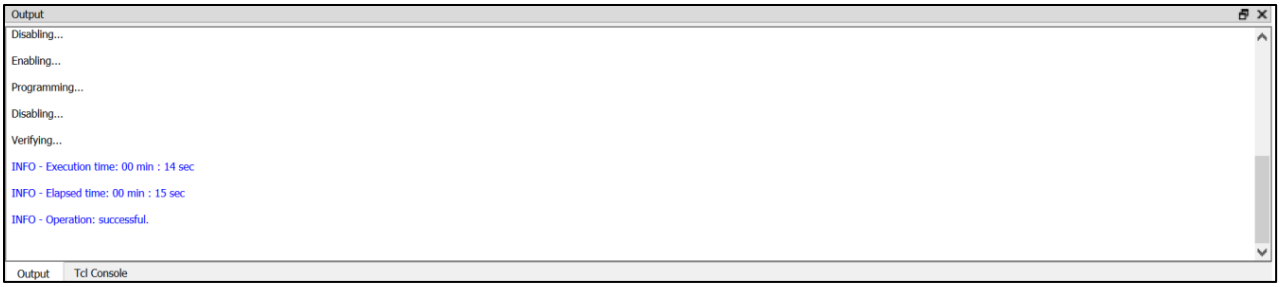

Figure 9.9. Lattice Radiant Programmer – Output Console

10. Running the Demo

To run the demo:

1. Power on the VVML board.
2. Make sure that the position of DIP SWITCH0 is ON to set FX3 to boot from I^2C EEPROM.
3. Flash the .bit and .mcs files.
4. Connect the VVML board to the PC through the USB3 port.
5. Open the AMCap video display application and select the FX3 Device as source from under **Devices**.
6. The camera image is displayed as shown in Figure 10.1.
7. Press the SW2 push button to register the Face ID, and the registered values can be cleared using the SW3 push button.

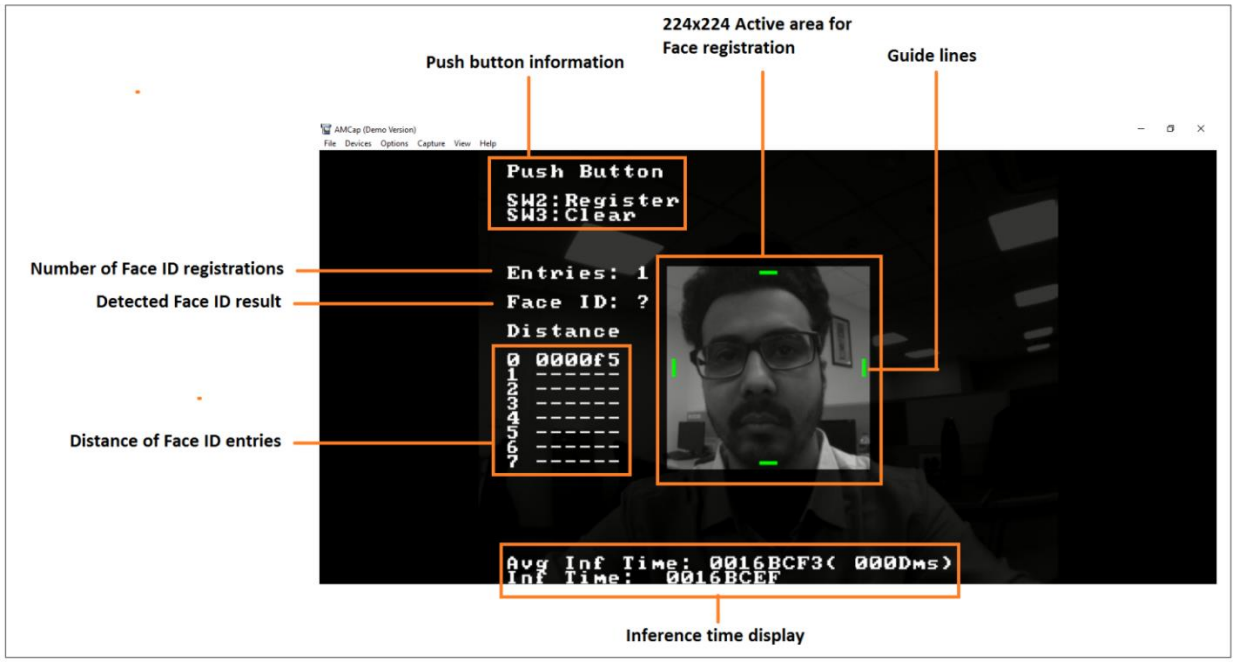

Figure 10.1. Demo Camera Image

If Inference time display is disabled, and .bit file is generated using *face_id_display.mem*. The text and values of Inference Time Display are not displayed in the output. The output shown in Figure 10.1 is with the .bit file having Inference time enabled using the *face_id_display_INF.mem* file.

10.1. Ideal Conditions for Testing the Demo

- Distance – The user's face should completely fit the guide lines.
- Lighting – Proper lighting is needed to efficiently run demo. Too low and direct light from a source may reduce the performance quality of the demo.

Appendix A. Other Labeling Tools

Table A.1 provides information on other labeling tools.

Table A.1. Other Labeling Tools

Software	Platform	License	Reference	Converts To	Notes
annotate-to-KITTI	Ubuntu/Windows (Python based utility)	No License (Open source GitHub project)	https://github.com/SaiPrajwal95/annotate-to-KITTI	KITTI	Python based CLI utility. Just clone it and launch. Simple and Powerful.
LabelBox	JavaScript, HTML, CSS, Python	Cloud or On-premise, some interfaces are Apache-2.0	https://www.labelbox.com/	json, csv, coco, voc	Web application
LabelMe	Perl, JavaScript, HTML, CSS, On Web	MIT License	http://labelme.csail.mit.edu/Release3.0/	xml	Converts only jpeg images
Dataturks	On web	Apache License 2.0	https://dataturks.com/	json	Converts to json format but creates single json file for all annotated images
LabelImg	ubuntu	OSI Approved :: MIT License	https://mlnotesblog.wordpress.com/2017/12/16/how-to-install-labelimg-in-ubuntu-16-04/	xml	Need to install dependencies given in reference
Dataset_annotator	Ubuntu	2018 George Mason University Permission is hereby granted, Free of charge	https://github.com/omenyayl/dataset-annotator	json	Need to install app_image and run it by changing permissions

References

- Google TensorFlow Object Detection GitHub
- Pretrained TensorFlow Model for Object Detection
- Python Sample Code for Custom Object Detection
- Train Model Using TensorFlow
- https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

For more information, refer to:

- CrossLink-NX Family Devices Web Page
- Lattice sensAI Solution Stack Web Page
- Lattice Radiant Software Web Page
- Lattice Insights for Training Series and Learning Plans

Technical Support Assistance

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

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

Revision History

Revision 1.1, October 2023

Section	Change Summary
All	Corrected the document number to *FPGA-RD-02244*.
Disclaimers	Updated this section.
References	Added links to the CrossLink-NX Family Devices Web Page, Lattice sensAI Solution Stack Web Page, Lattice Radiant Software Web Page, and Lattice Insights for Training Series and Learning Plans.
Technical Support Assistance	Added the link to Lattice Answer Database.

Revision 1.0, November 2021

Section	Change Summary
All	Initial release.

www.latticesemi.com