IMPLEMENTATION OF TRAFFIC SIGNS RECOGNITION ON INTEL CYCLONE V E DEVELOPMENT KIT FOR EMBEDDED SYSTEM APPLICATION

MUHAMMAD HANIF BIN ABU KASSIM

UNIVERSITI TEKNOLOGI MALAYSIA

PSZ 19:16 (Pind. 1/13)

UNIVERSITI TEKNOLOGI MALAYSIA

DECLARATION OF THESIS / UNDERGRADUATE PROJECT REPORT AND COPYRIGHT

Author's full name : Muhammad Hanif Bin Abu Kassim

Date of Birth : 28 August 1998

Title : Implementation Of Traffic Signs Recognition On Intel FPGA Cyclone VE SoC For Low Power Consumption In Embedded System Application

Academic Session : 2022/2023-2

I declare that this thesis is classified as:

☐ **CONFIDENTIAL** (Contains confidential information under the Official Secret Act 1972)*

☐ **RESTRICTED** (Contains restricted information as specified by the organization where research was done)*

☑ **OPEN ACCESS** I agree that my thesis to be published as online open access (full text)

1. I acknowledged that Universiti Teknologi Malaysia reserves the right as follows:
2. The thesis is the property of Universiti Teknologi Malaysia
3. The Library of Universiti Teknologi Malaysia has the right to make copies for the purpose of research only.
4. The Library has the right to make copies of the thesis for academic exchange.

	Certified by:
SIGNATURE OF STUDENT	**SIGNATURE OF SUPERVISOR**
A20MJ0060	Mdm. Nordinah Binti Ismail
MATRIC NUMBER	**NAME OF SUPERVISOR**
Date: 12 JANUARY 2024	Date: 12 JANUARY 2024

NOTES : If the thesis is CONFIDENTIAL or RESTRICTED, please attach with the letter from the organization with period and reasons for confidentiality or restriction

"I hereby declare that I have read this final year project report and in my opinion this final year project is sufficient in term of scope and quality for the award of the degree of Bachelor of Electronic Systems Engineering"

Signature : _____

Name of Supervisor I : Mdm. Nordinah Binti Ismail

Date : 14 JANUARY 2024

[THIS PAGE INTENTIONALLY LEFT BLANK]

IMPLEMENTATION OF TRAFFIC SIGNS RECOGNITION ON INTEL CYCLONE V E DEVELOPMENT KIT FOR EMBEDDED SYSTEM APPLICATION

MUHAMMAD HANIF BIN ABU KASSIM

A final year project report submitted in partial fulfilment of the
requirements for the award of the degree of
Bachelor of Electronic System Engineering

Malaysia-Japan International Institute of Technology
Universiti Teknologi Malaysia

JUNE 2023

DECLARATION

I declare that this final year project report entitled *"*Implementation of Traffic Sign Recognition on Intel FPGA Cyclone V E Development Kit for Embedded Systems Application*"* is the result of my own research except as cited in the references. The final year project report has not been accepted for any degree and is not concurrently submitted in the candidature of any other degree.

Signature : .....

Name : MUHAMMAD HANIF BIN ABU KASSIM

Date : 12 JANUARY 2024

DEDICATION

I dedicate my thesis to my parents, who have always supported me. They taught me tenacity, excellence, and lifelong learning. I'm appreciative for their support. This thesis honors their unfailing support and trust in me. Their support will always be appreciated.

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to my supervisor, Madam Nordinah Binti Ismail, for her invaluable guidance and support in completing my Final Year Project. Her expertise and mentorship have been instrumental throughout the entire project, providing me with valuable insights and direction.

I would also like to extend my thanks to my amazing friends who have been a constant source of motivation and support throughout this journey. Their encouragement and assistance have played a significant role in helping me complete both the project and the report.

I am truly grateful to all these individuals who have played a pivotal role in my academic and personal growth. Their guidance, motivation, and support have been indispensable, and I am indebted to them for their contributions to my success.

ABSTRACT

The Traffic Sign Recognition (TSR) system implemented on the Intel FPGA Cyclone V E Development Kit demonstrates a significant stride in intelligent transportation. The goal is to develop a TSR model that can effectively interpret road signs and align with the local context by recognizing Malaysian road signs that suitable for embedded system setup. However, the challenge lies in optimizing the algorithm for the FPGA's soft core processing capabilities, as it requires writing the code in hardware descriptive language (HDL) or C/C++ for compatibility. The methodology involves acquiring a Malaysian road sign dataset, implement the Keras CNN medol for the road sign classification, conducting testing, model conversion from Keras model to C source code using (tf2onnx and onnx2c) and deploy the algorithm on Nios II soft core processor on Cyclone V E Development Kit board. The research concludes that implementing TSR on Cyclone V E Development Kit requires a strong commitment and expertise to learn and finding a solution to execute the algorithm on the FPGA's softcore processor. Future research could build on this groundwork to extend the system's capabilities, aiming for comprehensive traffic sign recognition with real-time adaptability.

ABSTRAK

Sistem Pengenalan Tanda Trafik (TSR) yang dilaksanakan pada Kit Pembangunan Intel FPGA Cyclone V E menunjukkan kemajuan penting dalam pengangkutan pintar. Matlamatnya adalah untuk mengembangkan model TSR yang dapat mentafsir tanda jalan raya dengan berkesan dan selaras dengan konteks tempatan dengan mengenal pasti tanda-tanda jalan raya Malaysia yang sesuai untuk persediaan sistem terbenam. Walau bagaimanapun, cabarannya terletak pada pengoptimuman algoritma untuk kemampuan pemprosesan teras lembut FPGA, kerana ia memerlukan penulisan kod dalam bahasa deskriptif perkakasan (HDL) atau C/C++ untuk keserasian. Metodologi ini melibatkan pengambilan set data tanda jalan raya Malaysia, melaksanakan model CNN Keras untuk pengelasan tanda jalan raya, menjalankan ujian, penukaran model dari model Keras ke kod sumber C menggunakan (tf2onnx dan onnx2c) dan melancarkan algoritma pada pemproses teras lembut Nios II pada papan Kit Pembangunan Cyclone V E. Penyelidikan ini menyimpulkan bahawa pelaksanaan TSR pada Kit Pembangunan Cyclone V E memerlukan komitmen dan kepakaran yang kuat untuk belajar dan mencari penyelesaian untuk menjalankan algoritma pada pemproses teras lembut FPGA. Penyelidikan masa depan boleh membina atas kerja asas ini untuk mengembangkan kemampuan sistem, dengan tujuan untuk pengenalan tanda jalan raya yang menyeluruh dengan kebolehserasan masa nyata.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

TSR	-	Traffic Sign Recognitiom
HLS	-	High-Level Synthesis
FYP1	-	Final Year Project 1
FYP2	-	Final Year Project 2
FPGA	-	Field Programmable Gates Array
SoC	-	System on Chip
UTM	-	Universiti Teknologi Malaysia
HOG	-	Histogram of Gradient
ANN	-	Artificial Neural Network
CNN	-	Convolutional Neural Network
ADAS	-	Advance Driving Assistance systems
GTSRB	-	German Traffic Sign Recognition Benchmark
SVM	-	Support Vector Machine
BRIEF	-	Binary Robust Independent Elementary Features
SURF	-	Speeded-Up Robust Features
ONNX	-	Open Neural Network Exchange
MSER	-	Maximally Stable external Region -

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background Study

The automobile industry has made significant advances in the field of car electronics in the twenty-first century. Electronic systems are widely used, and they provide additional intelligence to automobiles. These intelligence technologies include Advanced Driver Assistance System technologies (ADAS). ADAS assists the driver in driving the vehicle and leads to increased road awareness. As a result, ADAS makes vehicles and roadways safer for both drivers and pedestrians [1]. One of the key component of advance driver assistance system is traffic signs recognition.

1.1.1 Traffic Sign Recognition

Traffic sign recognition (TSR) is a vital part in car autonomous driving car. TSR will support modern intelligent transportation systems (ITS) which is ADAS by minimizing the possibility of accident and increasing road safety. Passenger safety has become the primary interest of automobile industry since the number of people killed in traffic accident has tragically increased. Traffic signs recognition application will caution the driver and command or prohibit specific tasks upon detecting road signs to avoid accident due to lack of attention or visibility. As a result, an autonomous system for detecting and recognising traffic signs would be quite useful. This appears to be a simple process because panels adhere to a well-defined standard in terms of form, colour, size, and location on the road. However, due to several limits connected to the status of these signs, the reality is different: they can be partially obscured or not fully visible due to weather conditions such as rain and fog, or because of the shadow of trees, buildings, or other things. All these issues might lead to incorrect identification of traffic signs or confusion with comparable items. Those three distinct characteristics are utilised to develop effective detection and recognition algorithms [2].

To construct the TSR, it involves three level of process, which is detection, classification, and recognition. The most difficult task in traffic sign recognition is detecting traffic signs [3]. It recognises and localises the position of traffic signs in images. In [4] it shows several methods for detecting traffic signs that have been proposed by using colour segmentation, shape information, colour and shape information, colour and Haar feature, Histogram of oriented gradient (HOG), and maximally stable external region (MSER).

Most intelligent system applications need high-speed operations to accommodate real-time limitations. Although satisfactory result can be achieved by implementing these systems just through software implementation, hardware implementation is required especially for sophisticated and multi-technique-based processing procedures. To provide execution time acceleration and to meet the requirement of mobility in embedded system. [2], [5]

1.1.2 Traffic Sign Recognition in Embedded and Non-Embedded.

Traffic sign recognition (TSR) plays a pivotal role in autonomous driving, with varying implementations in embedded and non-embedded environments. In non-embedded environments, TSR typically occurs on standalone computers or servers, equipped with substantial computational power. This setting allows for the use of advanced software algorithms, including deep neural networks, and often leverages cloud computing or server-based resources for data processing. However, these systems face challenges such as latency issues due to data transfer between the traffic environment and the processing unit, rendering them less practical for real-time applications like autonomous driving. In contrast, embedded environments integrate TSR systems into compact, dedicated hardware like FPGAs or microcontrollers. This approach focuses on optimizing performance to meet real-time requirements within resource-constrained settings. It involves hardware acceleration techniques and efficient algorithm design to balance accuracy with processing capabilities. Embedded systems, therefore, offer significant advantages in real-time processing and reduced latency, essential for autonomous vehicles and advanced driver-assistance systems (ADAS). While non-embedded systems provide the luxury of more computational

resources for complex algorithm processing, embedded systems are indispensable for immediate, on-the-spot decision-making required in autonomous driving. The choice between these environments hinges on specific application requirements, such as real-time processing needs, availability of computational resources, and integration complexities within the system.

1.1.3 Intel Cyclone V E Development Kit

The Intel Cyclone V E Development Kit is a robust platform designed for FPGA applications. It's renowned for its balance between performance, power efficiency, and cost-effectiveness, making it an attractive choice for a wide range of applications. Its main advantage lies in its flexibility and the ability to handle parallel processing, which is critical for real-time applications. However, its complexity in programming and resource constraints can be challenging, particularly for developers new to FPGA architectures.[6]

Embedded within the Cyclone V E Development Kit is the Nios II soft core processor, a configurable, soft-core processor that enhances the FPGA's capabilities. This processor is particularly significant for custom embedded applications. It allows developers to tailor the processor architecture to their specific application needs, providing a balance between performance and resource utilization. The Nios II processor is integral for tasks that require efficient data processing and real-time operations.[6], [7]

Implementing traffic sign recognition (TSR) on the Cyclone V E Development Kit, leveraging the Nios II platform, offers significant benefits. The FPGA's ability to process tasks in parallel, coupled with the configurable nature of the Nios II processor, makes it well-suited for the high-speed, accurate processing required in TSR. This setup can efficiently handle the image processing demands of TSR, ensuring real-time performance essential in autonomous driving systems. The combination of these technologies provides a powerful, adaptable solution for advanced TSR applications.

In our research, we're leveraging the Nios II platform on the Intel Cyclone V E Development Kit to implement cutting-edge deep learning techniques using Convolutional Neural Networks (CNNs) for traffic sign recognition. This approach is

about harnessing the power of deep learning to push the boundaries of what's possible in real-time image recognition, all within the compact and efficient framework of the Nios II platform. This project explores the implementation of TSR on hardware by running the algorithm written in high-level language in the NIOS-II “soft” processor on the FPGA system board. The research can be carried out further where the algorithm would be translated and optimized in hardware descriptive language (e.g. Verilog or HDL) so that the implementation would be eventually on the hardware logic fabrics of the FPGA, namely Intel Cyclone VE Silicon-OnChip (SOC) board.

1.2 Problem Statement

Implementing a traffic sign recognition (TSR) system on autonomous vehicles is crucial for effectively interpreting road signs and providing valuable data to support self-driving capabilities. To meet the specific requirements of portability and task efficiency, it is necessary to develop the TSR system as an embedded system. In this regard, implementing the TSR algorithm on a Field-Programmable Gate Array (FPGA) platform offers a promising solution. However, implementing intelligent recognition task in embedded system has challenges compared to the desktop (non-embedded) environment where numerous of libraries and tools are available. directly transferring a software algorithm to an FPGA frequently yields unsatisfactory outcomes due to the optimization of many image processing algorithms for sequential processors. To implement it on Intel FPGA Cyclone V E Development Kit, it requires extra skill, knowledge and effort to successfully implement it. Furthermore, most of the research studies that related were implemented the traffic sign recognition on the Xilinx FPGA board[2], [3], [5], [8], [9].

Another challenge arises is the multi-variance of pattern, shape and colour in traffic sign are worldwide. Different countries may employ different patterns and designs for their road signs. To address this, our research might need to focus specifically on implementing the Malaysian road sign, aligning with the local context.

1.3 Research Goal

The aim of this research is the development and deployment of a Traffic Sign Recognition (TRS) system that operates efficiently on an FPGA platform, specifically tailored for Malaysian road signs.

1.3.1 Objective

The objectives delineated in this research are comprehensive and rooted in practical outcomes:

1. To implement a traffic sign recognition algorithm that leverages the capabilities of the Nios II soft-core processor, integrated with additional necessary hardware modules within the Intel FPGA Cyclone V E Development Kit. This includes the classification of Malaysian traffic signs, executed on the FPGA platform.
2. To refine the classification process for traffic signs, specifically Malaysian road signs, within the traffic sign recognition system. This includes the application of a Convolutional Neural Network (CNN) model that has been trained, tested, and validated to ensure high accuracy and efficiency in sign classification.

1.3.2 Scope

This research project aims to develop a software design for a traffic sign recognition system specifically focused on Malaysian road signs. The system will be implemented on the Intel FPGA Cyclone V E Development Kit using Nios II platform. The project targets four classes of road signs: stop signs, yield signs, no U-turn signs, and speed limit 30mp/h signs. The main objective is to classify and recognize these detected signs accurately.

The system design for this research project is structured to classify of the detected signs which recognition task. The proposed method will be influenced by utilizing existing literature to balance accuracy and processing time to ensure optimal

results. In this project, the target is to be able to implement the CNN trained model of the traffic sign recognition algorithm on Cyclone V E using Nios II platform.

By focusing on the specific requirements of Malaysian road signs and leveraging the capabilities of the Intel FPGA Cyclone V E Development Kit, this project aims to contribute to the field of traffic sign recognition and pave the way for more efficient and accurate systems in the future.

1.4 Project Significance

The major work and significance of this project lies in the successful integration of sophisticated classification algorithms developed in a desktop environment with their subsequent deployment in an FPGA environment. This dual-platform approach is pivotal to the project's contribution to the field of Traffic Sign Recognition (TSR).

By harnessing the computational strengths of a desktop environment for the development and refinement of AI deep learning algorithms, and then effectively transferring and optimizing these algorithms for the real-time constraints of an FPGA platform, this research bridges the gap between theoretical AI research and practical, real-world applications. The Nios II platform has been instrumental in illustrating the feasibility of this approach, showcasing the potential for FPGAs to address complex tasks within autonomous driving technologies and beyond.

The project extends its significance by providing a template for future research endeavors, demonstrating that the integration of development environments can lead to robust, efficient, and scalable solutions in embedded systems.

1.5 Thesis Outline

This research project is structured into five chapters, each of which focuses on discussing and explaining the necessary approaches to address the objectives and problem statement.

Chapter 1 provides an overview of the project, including the research background, problem statement, and research goals. It highlights the significance of the project in the context of traffic sign recognition.

Chapter 2 conducts a comprehensive review of the current challenges faced in implementing traffic sign recognition on the Intel FPGA Cyclone V E Development Kit. It examines various methods and techniques employed by previous projects that implemented TSR systems on different FPGA platforms. Additionally, this chapter discusses the limitations associated with TSR implementation relevant to this particular project.

Chapter 3 delves into the project's development, hardware and software that has been use. It also presents project implementation of the Traffic sign recognition system in hardware and software design.

Chapter 4 focuses on the results obtained and limitation. It discusses the reason behind all the output result to come out with the future work.

Lastly, in Chapter 5, the research concludes with an overall summary of the observations and accomplishments throughout the project. It also highlights potential future work that could be pursued to further enhance our implementation of the traffic sign recognition system on FPGA board.

CHAPTER 2

LITERATURE REVIEW

This chapter will discuss and review the related paper on the implementation of traffic sign recognition on FPGA and the intel Cyclone V E Development Kit FPGA itself. This chapter is essential when researching to gather information and knowledge about the project title.

2.1 Introduction

Traffic sign recognition (TSR) has garnered significant interest in recent years due to its importance in various applications, including autonomous driving, advanced driver assistance systems (ADAS) [10], and road safety improvements. TSR detects and classifies traffic signs, helping drivers and autonomous vehicles make decisions for safer roads.

TSR must tackle various difficulties, like changes in lighting, objects blocking the view, and a variety of sign types[11]. These challenges require robust algorithms and techniques to handle different scenarios effectively. Therefore, a comprehensive review of existing approaches and algorithms is necessary to identify the most suitable methods for implementing traffic sign recognition on the Intel FPGA Cyclone V E Development Kit for embedded system applications.

2.2 Existing Approaches and Algorithms

In traffic sign recognition, there are various vision-based algorithms and techniques that have been proposed in the literature. It is hard to compare or evaluate empirically their approaches since different databases, methods, and hardware/software platforms are used[8]. Moreover, certain studies were focusing on specific subsets of road signs, such as warning and speed limit signs. Several

approaches and algorithms have been developed to address the challenges posed by varying lighting conditions, occlusions, and different types of traffic signs. A review of these techniques provides insights into the strengths and limitations of different methods, aiding in the development of efficient and accurate traffic sign recognition systems.

2.2.1 Feature Extraction

One common approach in traffic sign recognition is feature extraction. Viola and Jones proposed the use of Haar-like features, which enable rapid object detection [12]. Another popular feature extraction method is the Histogram of Oriented Gradients (HOG) [13]. HOG captures the distribution of gradient orientations in an image, allowing for the detection of important features within traffic signs.

i. Haar and adaBoost Classifier

Haar features calculate adjacent rectangular rectangles within a detection window. Summing each region's pixel intensity and finding the differences in this calculation. By removing black pixels from white pixels, it quantifies the gradient shift from white to black. Haar feature calculation requires scanning several windows, which may delay training and detection. The integral image approach can calculate rectangular eigenvalues in real-time. Avoiding superfluous computations improves Haar feature computation with this method. Figure 2-1 and 2-2 illustrates Haard + adaBoost feature and implementation flow for traffic sign recognition.[14]

In [15], AdaBoost boosting technique helps train classifiers. Training many weak classifiers on the same data and merging them using weighted integration creates a robust classifier. AdaBoost improves classification by merging weak classifier outputs. Haar features calculate neighboring rectangles within a detection window. Eliminating needless computations using the integral image approach speeds computing. AdaBoost, a boosting method, trains and integrates numerous weak classifiers to generate a stronger, more accurate classifier. Figure 2-3 shows the outcome of of attempting to recognize traffic signs using Haar and AdaBoost.

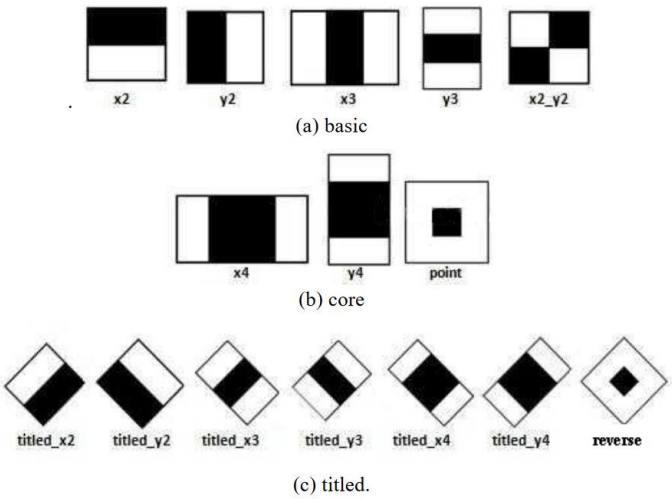


Figure 2-1 Haar rectangle feature.[14]

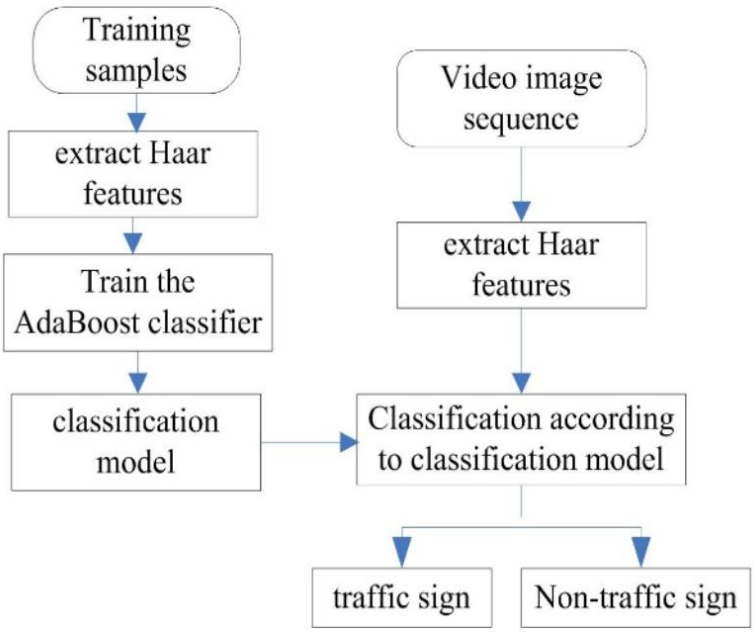


Figure 2-2 Traffic sign detection based on Haar + AdaBoost.[14]

Figure 4. Detection result

Figure 5. Detection result

Figure 2-3 The experimental outcome of attempting to recognize traffic signs using Haar and AdaBoost.[14]

ii. Histogram of Oriented Gradient, HOG

Histogram of Oriented Gradients (HOG) is an extensively employed feature extraction method for object detection in image processing. It operates under the premise that the appearance of an object can be described by its geometry. Each cell's histogram of oriented gradients is computed by considering the weighted contributions of each pixel's gradient value. Figure 2-4 shows the algorithms for calculating the values of both the horizontal and vertical gradients. Calculating the horizontal and vertical gradient values establishes the foundation for the HOG feature. HOG was initially developed to assist human object detection, but it has since become a popular and effective feature for automated object detection. Gradient orientations are typically quantified into bins during HOG feature extraction, with each bin representing a range of orientations. To account for variations in illumination, the image is divided into overlapping blocks, and histograms of gradient orientation are computed and normalized within each block. The histograms of all the blocks are combined to generate an exhaustive representation of the image's features. Figure 2-5 illustrates the conventional HOG feature extraction procedure. Due to their robustness against

variations in illumination and their capacity to manage local geometric and photometric transformations, HOG features are frequently incorporated into scene text recognition tasks. It is essential to observe, however, that HOG captures only the orientation of individual pixels, ignoring their spatial relationships. To increase accuracy and performance, HOG will come with Max-Margin Object Detection to make it simpler to implement and use since we can construct our own object detectors from a few samples of images. The procedure is fast enough to execute in real-time on a smart vehicle system. Figure 2-6 shows HOG extract traffic signs images .[16], [17]

$$dx = I(x+1, y) - i(x-1, y)$$

$$dy = I(x, y+1) - I(x, y-1)$$

Information:

dx = vertical image gradient value

dy = horizontal image gradient value

$I(x, y)$ = pixel value in row x and column y

Figure 2-4 The algorithms for calculating the values of both the horizontal and vertical gradients.[16]

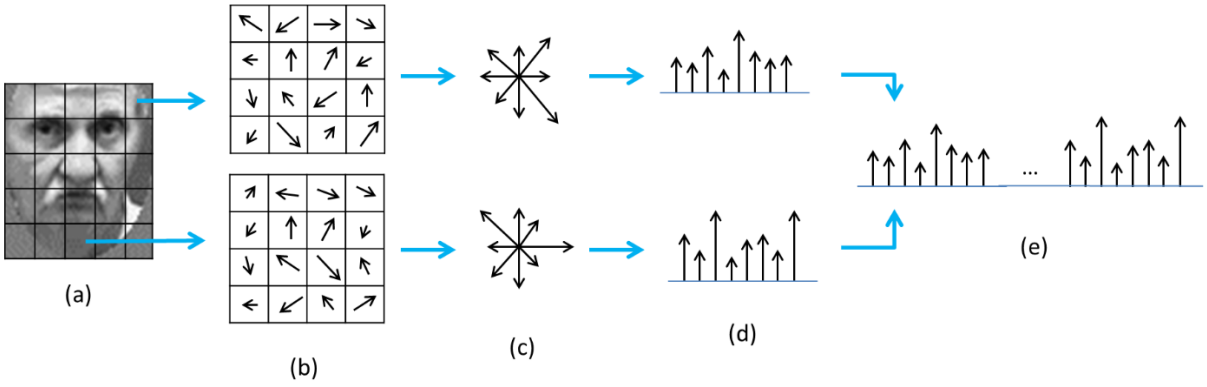


Figure 2-5 In order to extract HOG features from a picture, first the image is spatially segmented into cells, and then the orientation of each pixel inside a cell is calculated. Depending on the cell-space image division, histograms of successive orientations are generated and concatenated one after the other.[18]

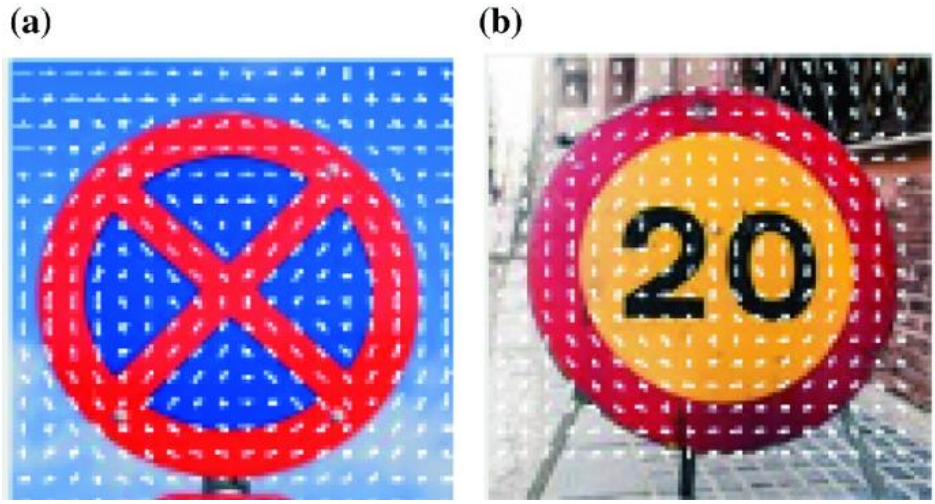


Figure 2-6 HOG extract traffic signs images.[19]

2.2.2 Machine Learning

Machine learning-based approaches have also proven effective in traffic sign recognition. Support Vector Machines (SVM) have been widely employed in this domain [20].SVMs are trained on labeled data to classify traffic signs based on extracted features. This technique focuses on identifying specific patterns and shapes that characterize traffic signs. SVMs use hyperplanes to divide classes. SVM seeks the hyperplane that minimizes error and increases class margin. SVM generates class-dividing hyperplanes iteratively. Figure 2-7 illustrates the SVM binary classification. SVM first iteratively creates these hyperplanes. Then, it chooses the optimum hyperplane to divide classes. Linear, polynomial, and radial basis function kernels improve SVM accuracy and strength. These kernels enable SVM's versatility. SVM excels in high-dimensional space, accuracy, and memory utilization. SVM has a long training time and cannot handle overlapping classes in big datasets. Random Forests, another machine learning algorithm, have also been used for traffic sign recognition tasks[21]. Random Forests excel at handling large and diverse datasets, making them suitable for this application. Figure 2-8 shows the classification accuracy that the random forest classifier achieves on average while using a variety of features.

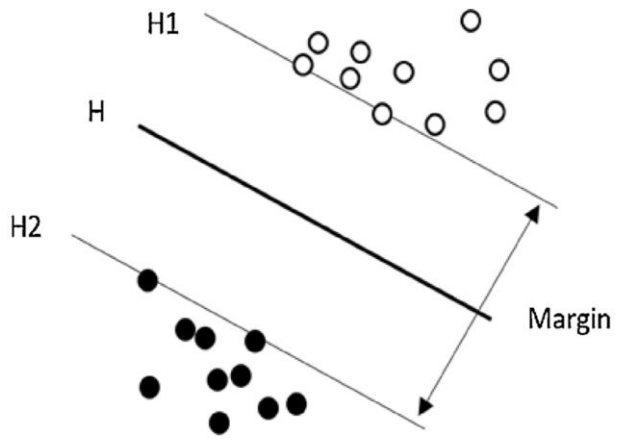


Figure 2-7 The SVM binary classification.[22]

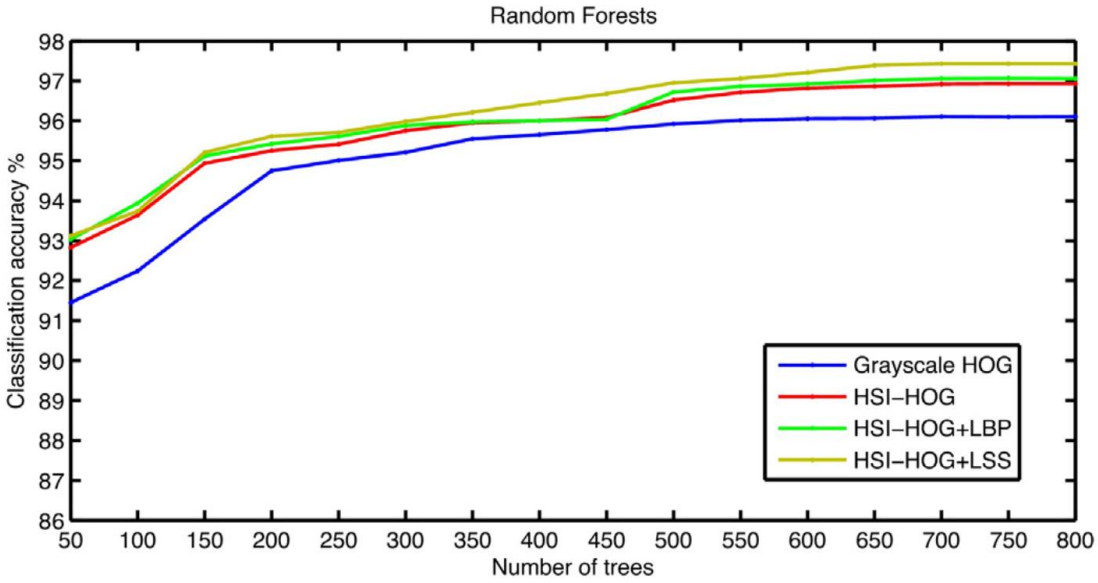


Figure 2-8 The classification accuracy that the random forest classifier achieves on average while using a variety of features.[22]

2.2.3 Deep Learning

Deep learning techniques, specifically Convolutional Neural Networks (CNN), have demonstrated remarkable performance in traffic sign recognition. CNNs leverage their ability to learn hierarchical representations from input data, allowing for automatic feature extraction and classification [23]. Region-based Convolutional Neural Networks (R-CNN) take this a step further by selectively examining regions of interest within an image, enhancing detection accuracy [24].

In [25] , the study presents a comprehensive analysis and experimental comparison of two prominent approaches: the color segmentation and convolutional neural network (C-CNN) approach and the fast region proposal convolutional neural network (Fast R-CNN) approach. Figure 2-9 illustrates a flowchart of the C-CNN process. In the C-CNN method, the input image undergoes color thresholding to select regions of interest (ROIs), effectively reducing the search space. Subsequently, a trained CNN is utilized to classify the ROIs based on whether they contain a traffic sign or not, followed by another CNN with the same architecture for traffic sign recognition. According to [26], the Fast R-CNN method incorporates various techniques to enhance training and testing speed, as well as improve detection accuracy. It achieves a remarkable 9x faster training than regular R-CNN [26] using the deep VGG16 network, and an impressive 213x faster testing time. To assess and compare the performance of the implemented approaches, the GERMAN TRAFFIC SIGN datasets were employed, specifically the GTSDB dataset for detection and the GTSRB dataset for recognition.

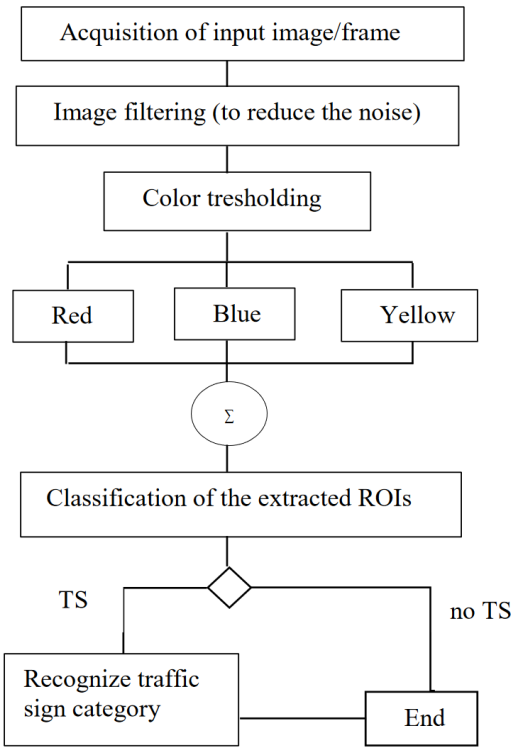


Figure 2-9 Flowchart of the C-CNN process.[25]

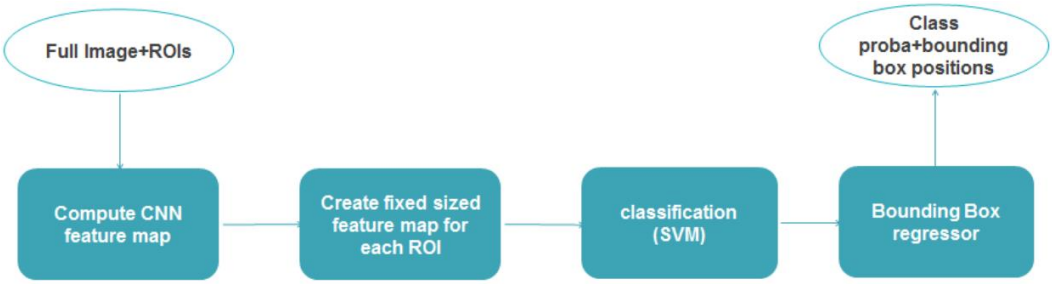


Figure 2-10 Flowchart of the Fast R-CNN architecture.[25]

2.3 Comparative Analysis of Traffic Sign Recognition Techniques

Table 2-1 Comparative Table of Existing Approaches and Algorithms for Traffic Sign Recognition (TSR)

Approach	Haar-like features and adaBoost Classifier	Histogram of Oriented Gradients (HOG)	Support Vector Machines (SVM) and Random Forests	Deep Learning (Convolutional Neural Networks - CNN)
Description	Utilizes Haar features for rapid object detection by calculating differences in pixel intensity within adjacent rectangular regions. Combined with AdaBoost for robust classification.	Captures the distribution of gradient orientations in an image for detecting traffic sign features.	SVMs classify traffic signs using hyperplanes based on extracted features. Random Forests handle large, diverse datasets for traffic sign recognition.	Leverages multi-layered neural networks for automatic feature extraction and classification. CNNs are highly effective in learning complex patterns in images, making them well-suited for recognizing and classifying traffic signs.
Pros	Effective for real-time calculations. Versatile for quick object detection.	Robust against variations in illumination. Effective for automated object detection.	SVM is accurate in high-dimensional space and efficient in memory usage. Random Forests are effective	High accuracy in feature detection and classification. Can automatically learn and improve from training data. Effective in handling various

			with complex datasets.	transformations of input images.
Cons	May involve delay in training and detection. Requires combination of weak classifiers for robustness.	Captures only the orientation of individual pixels, ignoring spatial relationships.	SVM has a long training time and struggles with overlapping classes in large datasets.	Requires large datasets and significant computational resources for training. Model interpretation can be challenging due to its 'black box' nature.
Applications in TSR	Used for initial detection of traffic signs by calculating gradient shifts	Utilized for extracting features from traffic signs to aid in their detection and classification.	Employed for classifying traffic signs based on patterns and shapes, especially in scenarios requiring specific identification.	Ideal for complex traffic sign recognition tasks where high accuracy and adaptability to various sign appearances are crucial. CNNs can effectively handle the classification of traffic signs into multiple categories and are particularly useful in scenarios involving diverse and extensive traffic sign datasets.

In my project, I implemented a Convolutional Neural Network (CNN) using Keras, which represents a sophisticated deep learning approach. The model is structured as a sequential model, incorporating layers such as convolutional, max pooling, dropout, flattening, and dense layers. This architecture is specifically designed to efficiently extract and learn distinctive features from traffic sign images, which is crucial for accurate classification.

2.4 Utilization of Keras and TensorFlow in CNN Model Development.

In the landscape of machine learning and particularly in the development of convolutional neural networks (CNNs), Keras and TensorFlow emerge as pivotal tools. Keras, a high-level neural networks API, is designed for human beings, not machines, prioritizing user experience and modularity. It serves as an interface for the TensorFlow library, allowing for an accessible and efficient approach to building and experimenting with different neural network architectures [27].

TensorFlow, developed by the Google Brain Team, is a comprehensive, open-source software library for machine learning. It provides a flexible foundation for research and development of state-of-the-art models, enabling the deployment of machine learning across a myriad of platforms, from servers to edge devices. It is particularly adept at facilitating the creation and training of deep neural networks with a scalable and distributed architecture [28].

The synergy between Keras and TensorFlow played a significant role in this project, allowing for the design of a sophisticated CNN model tailored for traffic sign recognition. The ease with which Keras allows for rapid prototyping, combined with TensorFlow's scalable deployment capabilities, made them an ideal choice for developing a model that is both accurate and performant on diverse platforms, including the desktop environment for training and the FPGA platform for execution.

2.5 FPGA-Based Image Processing

Figure 2-11 Illustrates a Cyclone V E FPGA Development Board Block Diagram. The Intel FPGA Cyclone V E Development Kit offers several features that enhance its power and efficiency for image processing. One such feature is the inclusion of the Nios II softcore processor. The Nios II processor allows for hardware/software co-design, enabling developers to implement complex image-processing algorithms in both hardware and software. This flexibility enables efficient utilization of resources and facilitates the integration of custom accelerators for computationally intensive tasks. In addition to the Nios II processor, the Intel FPGA Cyclone V E Development Kit incorporates other features that contribute to efficient image processing. The FPGA fabric itself provides a highly parallel processing

architecture, enabling the implementation of multiple image-processing tasks in parallel. This parallelism accelerates computation and improves overall processing speed.[6], [7], [29]

Furthermore, the Intel FPGA Cyclone V E Development Kit offers on-chip memory resources that can be leveraged for storing intermediate data and reducing external memory access. This on-chip memory helps minimize data transfer latency and enhances overall performance. Additionally, the FPGA fabric can be customized to include dedicated hardware accelerators, tailored specifically for image processing tasks. These accelerators offload computation from the processor, resulting in further speed improvements.

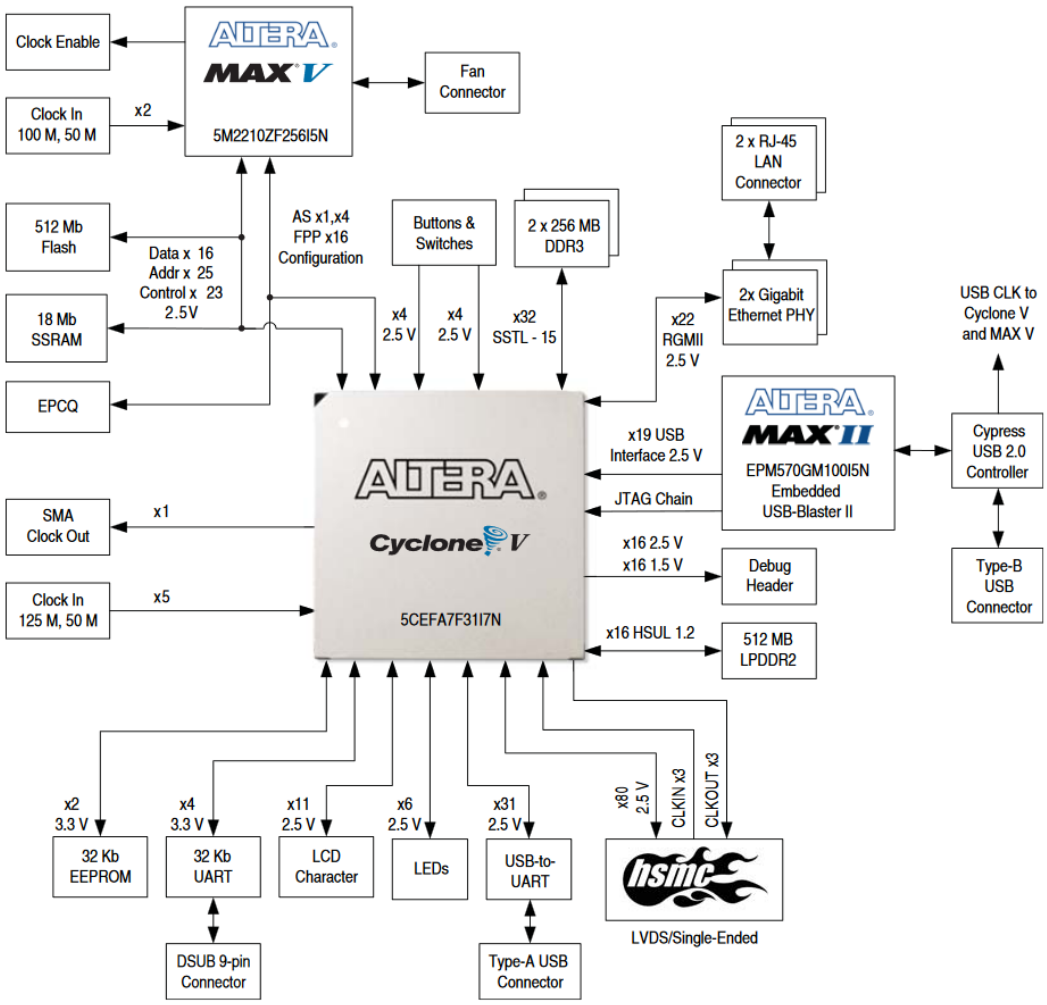


Figure 2-11 Cyclone V E FPGA Development Board Block Diagram[30].

By utilizing the Intel FPGA Cyclone V E Development Kit, researchers can harness the power and efficiency of its features, such as the Nios II processor, parallel FPGA fabric, on-chip memory, and hardware accelerators. These features facilitate hardware/software co-design, parallel processing, and optimized resource utilization, enabling the development of efficient and high-performance traffic sign recognition systems. This literature has proven how efficiently and the capabilities of image processing to be implemented on an FPGA board. But to utilize all that on FPGA, it required expertise and deep knowledge of both the hardware/software of FPGA and a strong understanding of image processing.

2.6 Image and Dataset

Traffic signs are an essential and vital component of the traffic world, and they can tell a lot about how traffic is going right now. They are made to control the flow of traffic, show drivers where dangers and problems are, give them tips, and help them find their way, which makes driving safe and easy. There are two main types of traffic signs: those that use ideograms and those that use words. The first group uses simple ideographs to show what the sign means. The second group uses words, lines, and other symbols to show what the sign means. Most study on traffic signs has been done on the first group. The most important types of road signs are those that warn, ban, require, or give information. Figure 2-12 shows some of the most popular road signs used in Malaysia. Different countries also may use different signs. In general, traffic signs are well-designed. They use certain colors (yellow, red, blue, white, and black) and shapes (triangle, rectangle, circle, and octagon) to stand out against the natural environment and help drivers recognize them.[31]

In the field of traffic sign recognition (TSR) systems, researchers have focused on developing algorithms and datasets to improve accuracy. Several public databases, such as the German Traffic Sign Recognition Benchmark (GTSRB) and the Belgium Traffic Sign Classification Benchmark (BTSCB), have been utilized for TSR research[8]. The paper [31] introduces a dataset specifically designed for the detection and recognition of Malaysian traffic signs (MTSD). This dataset aims to address the

gap in existing datasets by including a diverse range of traffic sign scenes. Unique features of MTSD include its collection under various conditions such as rain and night, and its incorporation of images from multiple sources, including mobile cameras and Google Maps. The dataset is designed to enhance the robustness of Traffic Sign Detection and Recognition (TSDR) techniques, particularly in challenging weather and lighting conditions. This dataset's diverse and comprehensive nature makes it a valuable resource for developing and testing advanced TSDR systems, particularly those focusing on Malaysian traffic signs.

Figure 2-12 Standard TS hierarchy illustrates the class, class ID, name, and amount of each traffic sign.[31]

CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter provide the methodology of this project which include project's development, software (training the TSR model, verify the model, conversion to hardware readable format), hardware (Cyclone V E Development Kit), and project implementation.

3.2 Project Development

Figure 3-1 illustrates the project development workflow. It begins with collecting traffic sign images dataset for recognition to be training in the CNN model. After collecting sufficient dataset, the second step was modelling our Keras CNN model and running the training of the Keras CNN model. Once the model is trained, it undergoes a testing and verification process to confirm its accuracy. The next step is to convert the Keras model into the ONNX format, which is again tested and verified for performance consistency. Subsequently, the ONNX model is translated into C source code, enabling integration with the embedded system environment. The C model is compiled and tested to ensure it functions correctly within the constraints of the target hardware.

Following this, the implementation phase begins, where the model is deployed on the Cyclone V E board by using Nios II platform. The final step is testing and verification of the complete application on the board to ensure operational capability and reliability.

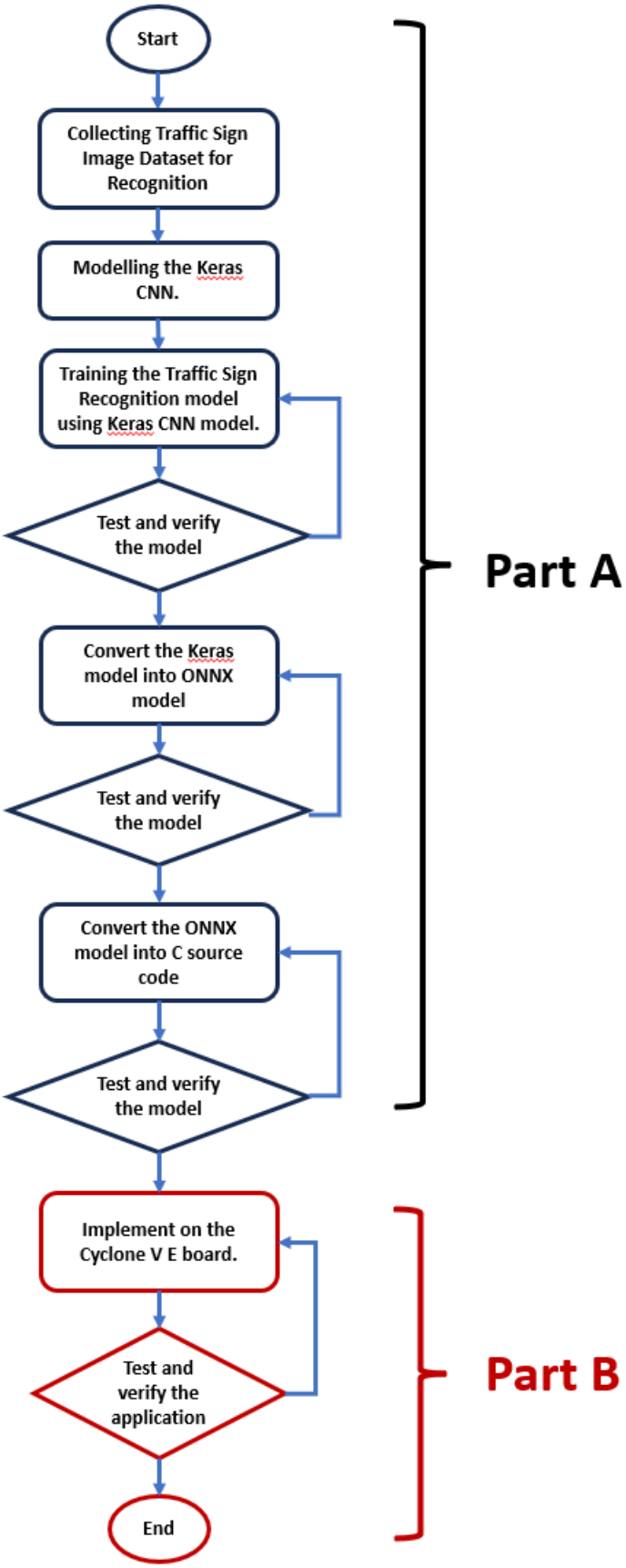


Figure 3-1 Workflow of the project development.

3.3 Hardware and Software

3.3.1 Cyclone V E Development kit

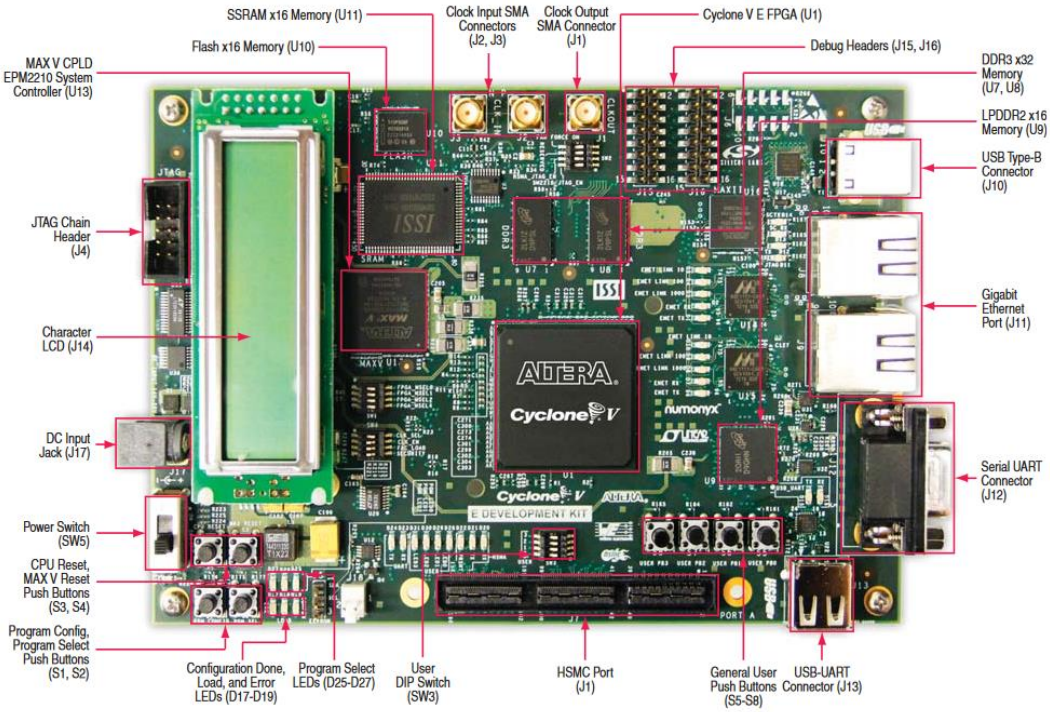


Figure 3-2 Cyclone V E FPGA Development Board[27].

Intel Cyclone V E FPGA development board is a compact and efficient platform ideal for developing low-power, high-performance applications. It is equipped with a rich suite of features including 149,500 Logic Elements, 56,480 Adaptive Logic Modules, ample memory resources such as DDR3 SDRAM, SSRAM, and LPDDR2 SDRAM, and a host of user interface components like LEDs and an LCD display [6]. Key to its embedded capabilities is the integration of the Nios II softcore processor, which offers customizable architecture to optimize performance and resource use. The board's compatibility with a variety of high-speed mezzanine cards (HSMCs) further extends its functionality, making it a versatile choice for intricate designs such as traffic sign recognition systems.

3.3.2 Quartus Prime Standard 23.1

Figure 3-3 Quartus software

Quartus Prime is a comprehensive FPGA software design suite from Intel (formerly Altera) that provides everything you need to design for Intel FPGAs, SoCs, and CPLDs. It offers a full design flow that includes project management, design entry, simulation, synthesis, optimization, and physical implementation. Quartus Prime is used for digital circuit design, verification, and to prepare a design for manufacturing.

In this project, the utilization of Intel's Quartus Prime Standard Edition 23.1 is specifically focused on designing the Nios II system within the platform designer and compiling it. Quartus Prime serves as an essential tool for configuring the Nios II processor and integrating it into the FPGA environment, a crucial step in setting up the foundation for the Traffic Sign Recognition (TSR) system. This process involves defining the necessary hardware components and ensuring their seamless interaction within the Nios II softcore CPU.

For the Quartus Prime installation and documentation, here below is the official link. It is a software package that include Nios II Embedded Design Suite:

- https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

3.3.3 Nios II Software Build Tools (SBT) for Eclipse

Figure 3-4 Nios II SBT for Eclispe software.

The Nios II Software Build Tools (SBT) for Eclipse is an integrated development environment (IDE) that extends the capabilities of Eclipse to support software development for the Nios II processor, a soft-core processor configurable for different hardware environments on Intel FPGA devices. Nios II SBT is used for writing, compiling, debugging, and managing applications that run on the Nios II processor.[29]

Nios II Software Build Tools (SBT) for Eclipse plays a key role in the software development phase of this project. Through this integrated development environment, we develop and implement the traffic sign recognition algorithm in C. The primary function of Nios II SBT for Eclipse in this project is to facilitate the coding, compilation, and debugging of the software that runs on the Nios II system, which have meticulously designed in Quartus Prime.[7]

3.3.4 Google Colab

Google Colab, an essential component of my project, is a free cloud service provided by Google that offers a robust platform for machine learning and data analysis. It's particularly valuable for its ability to run Python code in a web browser with no setup required, offering free access to computing resources including GPUs and TPUs.

In this Traffic Sign Recognition (TSR) project, Google Colab played a pivotal role, especially in the training of the Keras-based Convolutional Neural Network (CNN) model. The platform's seamless integration with Google Drive allowed for easy storage and access to the extensive dataset of traffic sign images. The training process was executed within the Colab environment, utilizing its computational resources to handle the intensive demands of training a deep learning model.

Furthermore, Colab was instrumental in the model conversion process. I utilized the platform to execute 'tf2onnx', a tool for converting Keras models to the ONNX (Open Neural Network Exchange) format. This conversion was a key step in making the model compatible for further translation into C source code through 'onnx2c', another tool employed within the Colab environment. These conversion

processes were critical in preparing the CNN model for deployment on the Nios II processor system, ensuring the model's compatibility and functionality within the FPGA-based hardware setup of this project.

3.4 Project Implementation

3.4.1 Collecting Traffic Sign Image Dataset for Recognition

In the context of my Traffic Sign Recognition (TSR) project, the focus was narrowed down to four key traffic sign classes: yield, stop, 30 mp/h, and No U-turn. To create a comprehensive dataset, this project employed a two-pronged approach.

Firstly, utilized images from the Malaysian Traffic Sign Dataset (MTSD) detailed in the paper referenced [31]. The MTSD offers a diverse collection of traffic sign images that are highly representative of real-world scenarios. These images, which are meticulously categorized and annotated, provide a valuable resource for training recognition algorithms. The dataset includes a variety of sign scenes, captured under different conditions, ensuring that the trained model can adapt to various real-world environments. In addition to utilizing the MTSD, we started on gathering custom images by captured the real traffic signs. Capturing images of traffic signs in various real-life scenarios and environments, ensuring a variety of lighting conditions and angles. These images were then edited and processed to fit the training requirements of the TSR model.

The combined use of the MTSD and our own custom-collected images ensured a robust and diverse dataset. This dataset forms the training process for the TSR system, providing the necessary variety and volume of images to effectively train the CNN model. The diversity of the images, both in terms of types of signs and the conditions under which they were captured, is crucial for developing a TSR system that is accurate, reliable, and adaptable to different real-world scenarios.

3.4.2 Modeling and Training the Traffic Sign Recognition with CNN Keras

This project leveraged Google Colab, a powerful cloud-based environment, to train the Traffic Sign Recognition (TSR) model using a Convolutional Neural Network

(CNN) developed in Keras. This model was meticulously designed and structured to accurately identify and classify four essential traffic sign classes: yield, stop, 30 mp/h, and No U-turn.

Figure 3-5 show the model architecture was composed of multiple layers, and each layer playing a vital role in the learning process:

1. **Convolutional Layers:** The first layer, a Conv2D layer with 30 filters of size 5x5, was employed for initial feature extraction. This was followed by another Conv2D layer with 15 filters of size 3x3, further refining the features.

2. **MaxPooling Layers:** Post convolution, MaxPooling2D layers were used to reduce the spatial dimensions, helping in reducing the computational load and overfitting.

3. **Dropout Layer:** A dropout layer was included to prevent overfitting by randomly setting a fraction of input units to zero during training.

4. **Flatten and Dense Layers:** The Flatten layer transformed the 2D matrix data to a vector, allowing it to be fed into Dense layers for classification. The model included Dense layers with 128 and 50 neurons, respectively, and a final Dense layer with 4 neurons corresponding to the four traffic sign classes, using a softmax activation function for multi-class classification.

```
def larger_model():
    model = Sequential()
    model.add(Conv2D(30, (5,5), input_shape=input_shape, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Conv2D(15, (3,3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50,activation='relu'))
    model.add(Dense(4, activation='softmax'))
    return model

# build the model
model = larger_model()
```

Figure 3-5 The architecture of the Convolutional Neural Network (CNN) model for the Traffic Sign Recognition project.

The training process involved using an `ImageDataGenerator` for data augmentation, which is crucial for enhancing the model's ability to generalize. This included configurations like rescaling, shear range, zoom range, and horizontal flip. The dataset was divided into training and validation sets, with the model being trained over 100 epochs and a batch size of 20.

```
#Model's Parameter
img_width, img_height = 32, 32 # dimensions of our images.
nb_train_samples = 509
nb_validation_samples = 497
epochs = 100
batch_size = 20
dim = 3
```

Figure 3-6 Model's parameter

Throughout the training, the model's performance was evaluated based on accuracy and loss metrics, providing insights into its learning efficacy, and guiding further optimization. The final stage involved testing the model with a separate set of images to assess its real-world applicability and reliability in accurately predicting traffic sign categories. This rigorous training and validation approach ensured the development of a robust and reliable TSR model, capable of effective real-world deployment.

```
[ ]  # Specify the directory containing the images
     directory = '/content/drive/MyDrive/TSR/Test'

     # Get a list of all the image files in the directory
     image_files = [f for f in os.listdir(directory) if f.endswith('.jpg') or f.endswith('.png')]

     # Iterate over the image files and perform prediction
     for filename in image_files:
         filepath = os.path.join(directory, filename)

         sign = cv2.imread(filepath)
         cv2_imshow(sign)

         # Read and preprocess the image
         im = cv2.imread(filepath)
         im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
         im = cv2.resize(im, (img_width, img_height))
         im = np.reshape(im, [1, img_width, img_height, dim])

         # Make the prediction
         predicted_probabilities = model.predict(im)
         predicted_labels = np.argmax(predicted_probabilities, axis=1)

         # Display the prediction result
         if predicted_labels == 0:
             print(f"{filename}: Predicted as Yield")
         elif predicted_labels == 1:
             print(f"{filename}: Predicted as No Uturn")
         elif predicted_labels == 2:
             print(f"{filename}: Predicted as 30km/h")
         elif predicted_labels == 3:
             print(f"{filename}: Predicted as Stop")
         else:
             print(f"{filename}: Unable to determine the prediction")
```

Figure 3-7 Testing the model with a separate set of images from training and validation dataset.

3.4.3 CNN Model Conversion

The model conversion process in this Traffic Sign Recognition project is a critical step to transition the model from a research and development environment to a deployable format suitable for the Intel Cyclone V E FPGA using Nios II platform. This process is divided into two main sub-stages.

3.4.3.1 Keras to ONNX Model

Figure 3-8 ONNX Framework.

After successfully training and validating the CNN model in Keras, the first step in model conversion is translating it into the Open Neural Network Exchange (ONNX) format. This conversion is crucial because ONNX provides an open-source format that allows AI models to be used across various platforms, ensuring interoperability and flexibility [32]. Figure 3-9 show the command line to install 'tf2onnx' libraries.

```
[19] !pip install tf2onnx
```

Figure 3-9 Setup the Colab environment with 'tf2onnx'.

For this task, it utilized the 'tf2onnx' tool, a Python-based utility that facilitates the conversion of TensorFlow and Keras models to the ONNX format. By using 'tf2onnx', the model's architecture, along with its trained weights and biases, were seamlessly converted into a format recognized by the ONNX standard. This step was performed within the Google Colab environment, leveraging its computing resources to handle the conversion process efficiently. Figure 3-10 show how to perform the conversion of Keras into ONNX model.

```
# Import tf2onnx
import tf2onnx

# 'model' is my Keras model that has been defined and trained

# Convert Keras model to ONNX format
onnx_model, _ = tf2onnx.convert.from_keras(model)

# Save the ONNX model to a file
onnx_filename = '/content/drive/MyDrive/Colab Notebooks/TSR_phase1.onnx'
with open(onnx_filename, "wb") as f:
    f.write(onnx_model.SerializeToString())
```

Figure 3-10 How to perform the conversion of Keras into ONNX model.

The successful conversion to ONNX was a significant milestone, as it marked the readiness of the model for further translation into a format compatible with the embedded system on the Intel Cyclone V E FPGA.

3.4.3.2 ONNX to C Source Code

The next phase involved converting the ONNX model into C source code, a necessary step for deploying the model on the Intel Cyclone V E FPGA board. For this, it employed 'onnx2c', a specialized tool designed to translate ONNX models into C code. This conversion process is crucial for integrating the model with the embedded system on the FPGA, allowing the model to run directly on the Nios II softcore processor. Figure 3-11 show the commands line to setup ‘onnx2c’ libraries.

```
[ ]  !git clone https://github.com/kraiskil/onnx2c.git

[ ]  !cd onnx2c && git submodule update --init

[ ]  !mkdir -p onnx2c/build
     !cd onnx2c/build && cmake -DCMAKE_BUILD_TYPE=Release .. && make
```

Figure 3-11 Setup the environment for 'onnx2c' tool.

The 'onnx2c' tool translates the ONNX model into a C representation, preserving the structure and functionality of the original Keras model. The generated C code is then optimized for execution efficiency and compatibility with the Cyclone V E FPGA’s processing capabilities. Figure 3-12 show the command to perform the 'onnx2c' conversion.

```
!onnx2c/build/onnx2c "/content/drive/MyDrive/Colab Notebooks/TSR_phase1.onnx" > "/content/drive/MyDrive/model.c"

!cat /content/drive/My\ Drive/model.c
```

Figure 3-12 Command to perform the 'onnx2c' conversion.

Through these conversion steps, the Traffic Sign Recognition model was successfully transformed from a high-level Keras model to a deployable C code format, ready for integration and execution on the FPGA hardware. This dual-stage conversion process ensures that the sophisticated capabilities of the CNN model are

retained and effectively utilized in the hardware-based application of traffic sign recognition.

3.4.4 Implement TSR Algorithm on Intel Cyclone V E

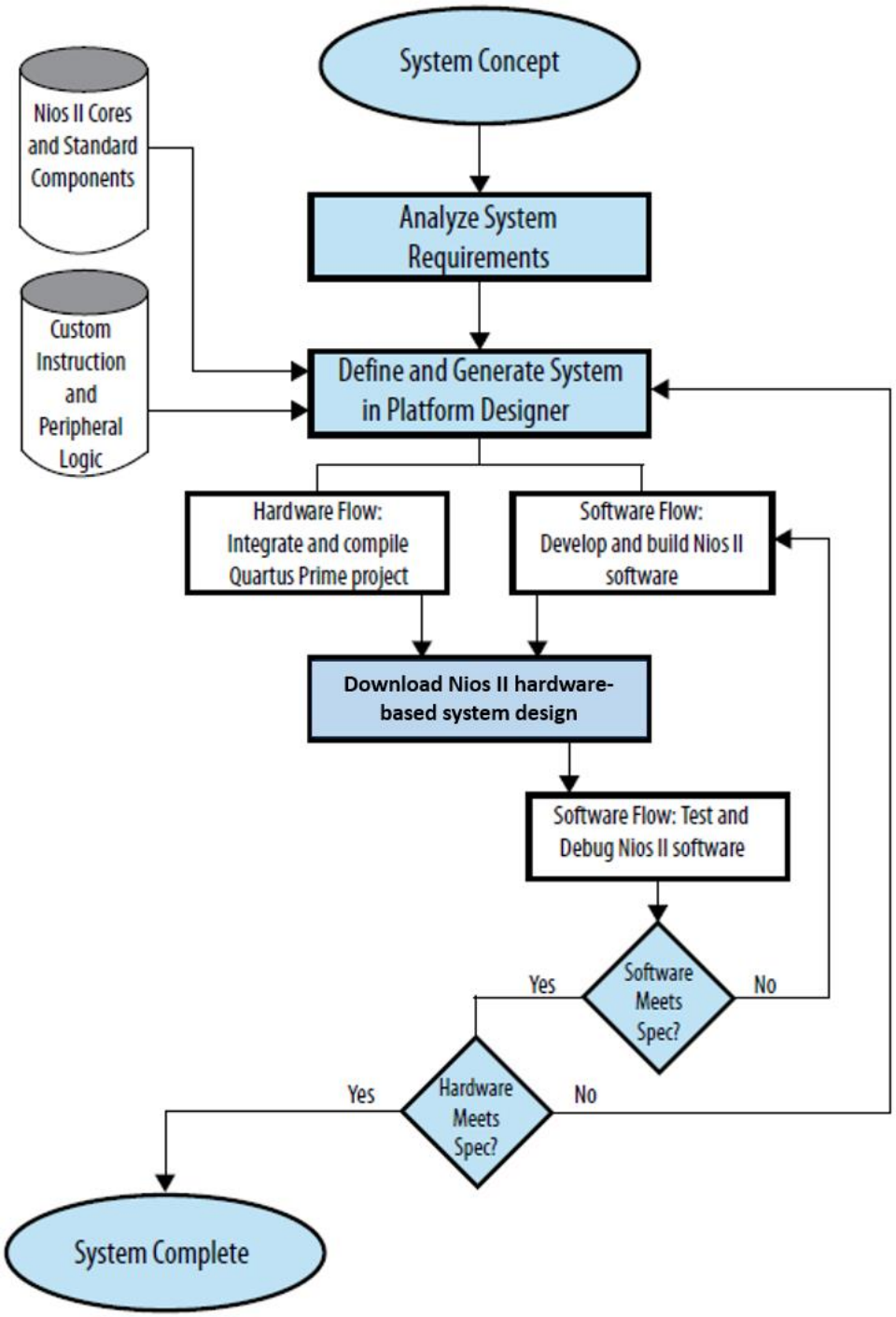


Figure 3-13 Implement TSR Algorithm on Intel Cyclone V E Nios II System Design Flow.

The final step of my project involves the actual deployment of the Traffic Sign Recognition (TSR) algorithm onto the Intel Cyclone V E FPGA. Figure 3-13 illustrates the implement of TSR Algorithm on Intel Cyclone V E Nios II System Design Flow. This stage is crucial as it translates all the preparatory work into a functioning system within the hardware. The process is divided into hardware and software design phases, which are iteratively refined to meet system specifications.

The hardware design phase is executed using Quartus Prime, where the Nios II processor system is defined and generated using the Platform Designer tool. The design includes Nios II cores, standard components, and any custom instruction or peripheral logic needed for the project. The Quartus Prime software is then used to synthesize the design, ensuring that our hardware of Nios II system design can perform the TSR algorithm. Once the design is integrated and compiled within Quartus Prime, it is downloaded to the target board which is on Intel Cyclone V E Development Kit.

Simultaneously, the software design is handled using the Nios II Software Build Tools (SBT) for Eclipse. This environment is where the software that runs on the Nios II processor is developed and built. It includes writing the code that implements the TSR algorithm, which has been converted into a C source format in the previous stages of the project. The software development phase involves testing and debugging to ensure the TSR algorithm operates correctly within the software constraints and meets the project's specifications.

The overall process, as depicted in the flowchart, outlines a systematic approach to implementing the TSR algorithm. It starts with the system concept and requirement analysis, leading to the definition and generation of the system in the hardware and software domains. Following the implementation on the hardware board and software development, iterative testing and debugging are conducted to ensure both the hardware and software meet the specified criteria. Only when both the hardware and software are verified to meet the necessary specifications is the system considered complete.

3.4.4.1 Hardware Design

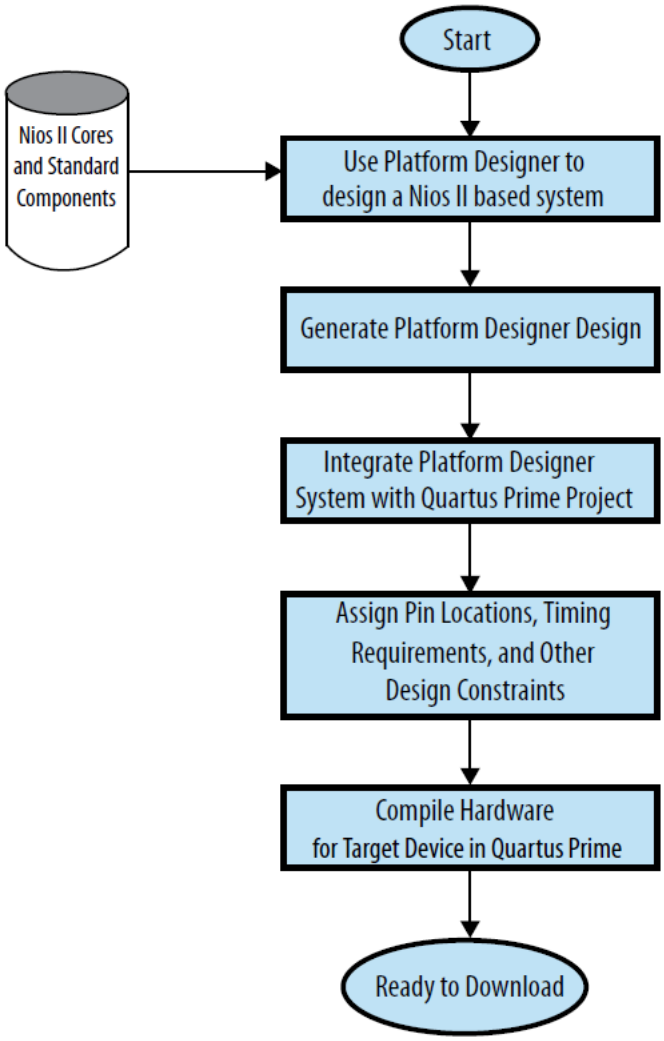


Figure 3-14 Nios II System Hardware Design Flow

Figure 3-14 illustrates Nios II System Hardware Design Flow. The hardware design for the Traffic Sign Recognition (TSR) project, executed using Quartus Prime, involved creating a Nios II system that serves as the main CPU for the application. The design was carefully architected to establish a balance between performance and resource utilization, tailored specifically for the Cyclone V E FPGA. The system was built around the Nios II processor, selected for its versatility and the extensive support it offers through its integration with Quartus Prime. It was connected to essential components such as on-chip memory, JTAG UART, switches, and an LCD, all interfaced through the Avalon bus a key interconnect framework that allows for modular and scalable designs.

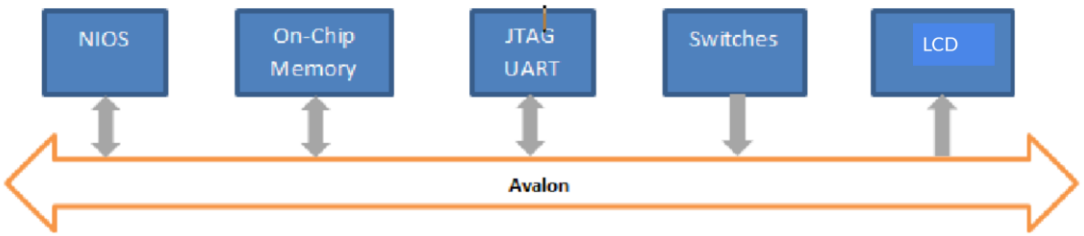


Figure 3-15 Nios II based system used in this project.

Figure 3-15 illustrates how Nios II based system used in this project. The hardware components were interconnected through the Avalon bus, an Intel specification that standardizes the communication interface between these components and the Nios II processor. This bus architecture simplifies design processes by enabling a common communication framework within the Cylone V E FPGA.

The Nios II core serves as the central unit of the system, managing computations and interfacing directly with the on-chip memory where the TSR algorithm's computational data resides. The JTAG UART module is essential for system diagnostics and communication with external devices, while user switches provide interactive control capabilities, and the LCD module is designated for real-time result display.

Following the integration of these components within the Platform Designer, the complete system design was generated. This design was then synthesized and compiled within Quartus Prime, taking into consideration pin locations, timing requirements, and other design constraints critical for the FPGA's performance. The meticulous compilation process ensured that the hardware design was optimized for the target Cyclone V E FPGA device. Upon successful compilation, the FPGA design was ready to be downloaded to the target board.

3.4.4.1.1 Components

3.4.4.1.1.1 Nios II processor

Nios II processor was chosen for its configurability and the ability to include custom instructions to accelerate specific operations within the TSR algorithm. The processor's specifications were optimized to balance speed and logic utilization, ensuring efficient use of the FPGA resources.

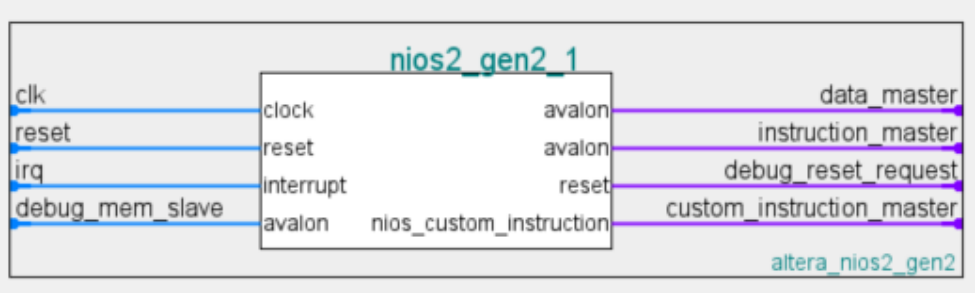
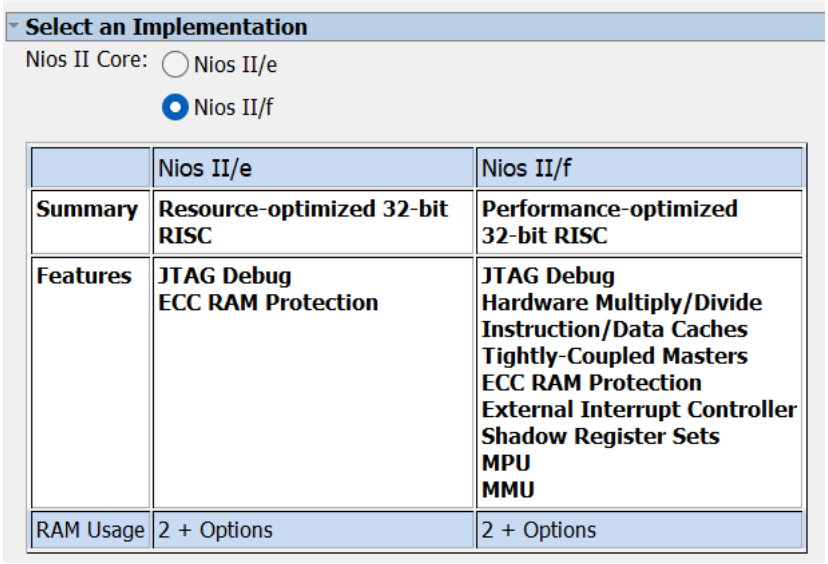


Figure 3-16 Nios II embedded processor block diagram.



	Nios II/e	Nios II/f
Summary	**Resource-optimized 32-bit RISC**	**Performance-optimized 32-bit RISC**
Features	**JTAG Debug** **ECC RAM Protection**	**JTAG Debug** **Hardware Multiply/Divide** **Instruction/Data Caches** **Tightly-Coupled Masters** **ECC RAM Protection** **External Interrupt Controller** **Shadow Register Sets** **MPU** **MMU**
RAM Usage	2 + Options	2 + Options

Figure 3-17 Configuring the Nios II as Nios II/f which is a performance type.

Figure 3-16 and 3-17 shows our Nios II embedded processor block diagram and how the configuration of Nios II setup in this project.

Nios II/f Processor:

- **Type**: Performance-optimized 32-bit RISC architecture, selected for its high performance.
- **Features**:
  - JTAG Debug: Essential for real-time debugging during development.
  - Hardware Multiply/Divide: Accelerates arithmetic operations, crucial for processing TSR algorithms.
  - Instruction/Data Caches: Increases the efficiency of data retrieval for the processor.
  - Tightly-Coupled Masters: Ensures fast and direct memory access, critical for performance.

- ECC RAM Protection: Protects data integrity, enhancing system reliability.
- External Interrupt Controller: Manages external events efficiently.
- Shadow Register Sets: Improves context switching performance, useful in multi-threaded scenarios.
- MPU (Memory Protection Unit): Provides security and prevents unauthorized memory access.
- MMU (Memory Management Unit): Facilitates advanced memory management capabilities.

The configuration of the Nios II/f processor within the Quartus Prime environment was further refined by specifying the reset and exception vectors, aligning with the on-chip memory, and ensuring that the processor starts execution correctly after reset or during exceptions.

The memory and peripherals associated with the Nios II/f processor were tailored to the project's needs:

- **On-Chip Memory:**
  - Configured with a size that matches the needs of the TSR algorithm, with ECC RAM Protection enabled for data integrity.
  - Initialization parameters were set up to ensure the memory content is loaded correctly on system start.
- **Instruction and Data Caches:**
  - Sized appropriately to balance memory usage with the performance benefits that caching provides.
- **Arithmetic Instructions:**
  - The arithmetic capabilities of the processor were optimized with the inclusion of a 32-bit multiplier for efficient computation.
  - Shift and rotate operations were set up to utilize logic elements (pipelined) for high-speed data manipulation.

The Nios II/f processor's ability to interface with the Avalon bus system allowed for smooth communication with other peripherals, such as JTAG UART for debugging, switches for user inputs, and LCD for displaying outputs, completing a

cohesive system optimized for our Nios II based system environment on Cyclone V E FPGA.

3.4.4.1.1.2 On-Chip Memory

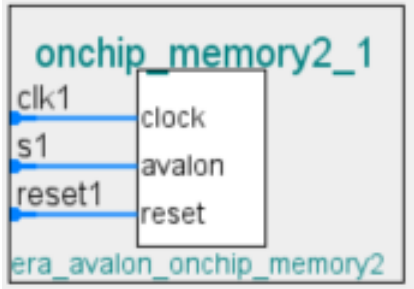


Figure 3-18 On-Chip Memory block diagram

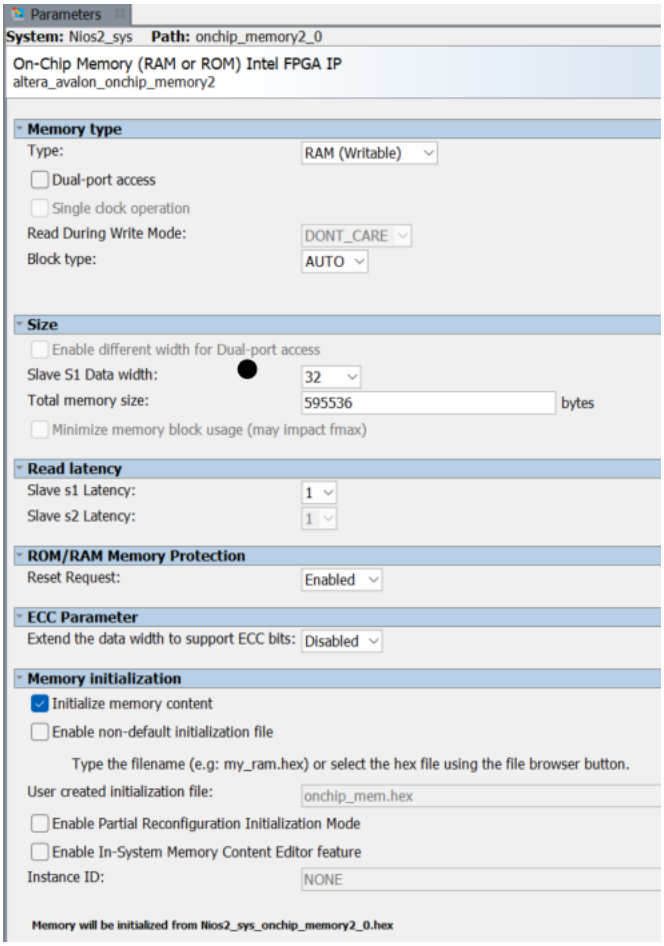


Figure 3-19 On-Chip Memory setup.

Figure 3-18 and 3-19 shows our On-Chip memory block diagram and how the configuration of on-ship memory was setup in this project. The on-chip memory of the Nios II system, as defined in the Traffic Sign Recognition (TSR) project, is a critical

component for the storage and retrieval of instructions and data. The configuration of this memory, as depicted as follows:

- **Dual-port Access**: This feature allows the system to perform read and write operations simultaneously, which is crucial for the efficient execution of the TSR algorithm. Dual-port access ensures that the processor can retrieve instructions while simultaneously writing data, or vice versa, without any access contention.
- **Size and Data Width**: The memory is configured with a total size of 595536 bytes and a data width of 32 bits for the Slave S1 port. This size is sufficient to store the TSR algorithm's code and the necessary data sets, while the 32-bit data width is aligned with the processor's data path, allowing for efficient data handling.
- **Read Latency**: The read latency for both Slave 1 and Slave 2 is set to 1, which reflects a balance between speed and the system's capability to handle data throughput without delays that could impact the real-time processing performance of the TSR application.
- **Memory Protection and Initialization**:
    - **ROM/RAM Memory Protection**: The Reset Request feature is enabled, ensuring that the memory can be reliably reset to a known state upon system startup or during error handling procedures.
    - **ECC Parameter**: The memory does not require Error-Correcting Code (ECC) support, as the application's tolerance for data errors and the memory's inherent reliability do not necessitate such protection.
    - **Memory Initialization**: The on-chip memory is initialized with predefined content from an initialization file (onchip_mem.hex). This ensures that upon system power-up or reset, the memory contains the correct instruction set and data for the TSR algorithm to start functioning immediately.

This memory configuration within the Quartus Prime environment was meticulously set up to ensure that the Nios II/f processor has efficient, reliable access to the necessary resources, enabling the real-time capabilities required for the Traffic Sign Recognition system.

.

3.4.4.1.1.3 JTAG UART

Figure 3-20 and 3-21 shows our JTAG UART block diagram and how the configuration of JTAG UART was setup in this project. The JTAG UART component in the Traffic Sign Recognition (TSR) project serves as a critical interface for debugging and communication between the development environment and the FPGA. This interface allows for data to be sent and received over the JTAG connection without requiring additional hardware like a UART-to-USB converter.

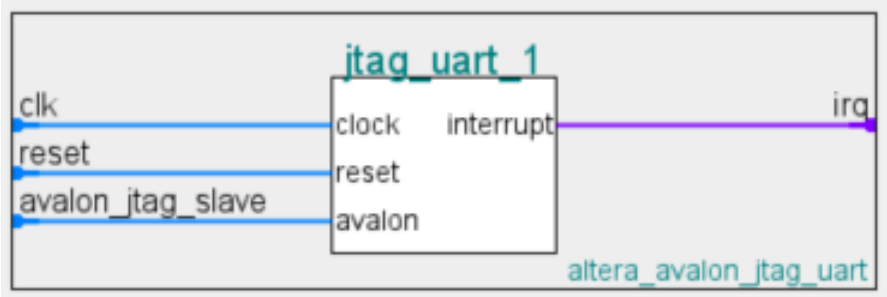


Figure 3-20 JTAG UART block diagram.

- **Block Diagram Interface:**
  - The JTAG UART IP core is connected to the system clock (`clk`) and reset (`reset`) signals.
  - The `avalon_jtag_slave` interface connects the core to the Avalon system bus, facilitating communication with the Nios II processor.
  - An interrupt line (`irq`) is provided to signal the Nios II processor when data is received or when the transmit buffer is empty.

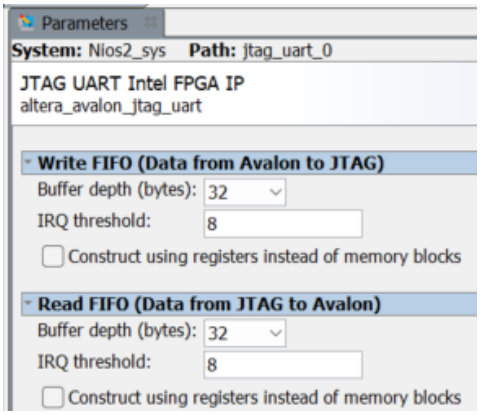


Figure 3-21 JTAG UART setup.

- **FIFO Buffers Configuration:**
  - Both the Write FIFO (data from the Avalon bus to JTAG) and Read FIFO (data from JTAG to the Avalon bus) are set with a buffer depth of 32 bytes. This size is adequate for handling the typical data packets expected during debugging sessions.
  - The IRQ (Interrupt Request) threshold is set to 8, meaning that an interrupt will be triggered when the FIFO reaches 8 bytes of data. This helps in managing data flow, ensuring the processor is notified in a timely manner to handle incoming or outgoing data.

By configuring the JTAG UART with these settings, the system can maintain efficient communication with the development environment, providing a reliable channel for program download, debugging output, and interactive control during development and testing of the TSR system. The JTAG UART is thus an integral part of the hardware design, ensuring that developers can easily monitor and interact with the system as it operates.

3.4.4.1.1.4 LCD

Figure 3-22 and 3-23 shows our LCD PIO block diagram and how the configuration of LCD PIO was setup in this project. The LCD display in the Traffic Sign Recognition (TSR) project is interfaced using a Parallel I/O (PIO) component within the Quartus Prime environment. Here's how the LCD and its interfacing PIO are configured in design platform:

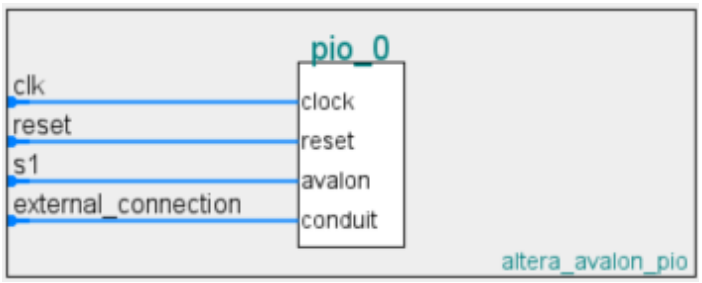


Figure 3-22 LCD PIO block diagram.

- **PIO Block Diagram Interface:**
  - The PIO is connected to the system clock (clk) and reset (reset) signals to ensure synchronous operation with the system.
  - The s1 signal serves as the data bus for the LCD, transmitting control and data signals to the display.
  - An external_connection conduit is used to facilitate connections to external hardware, such as the LCD.

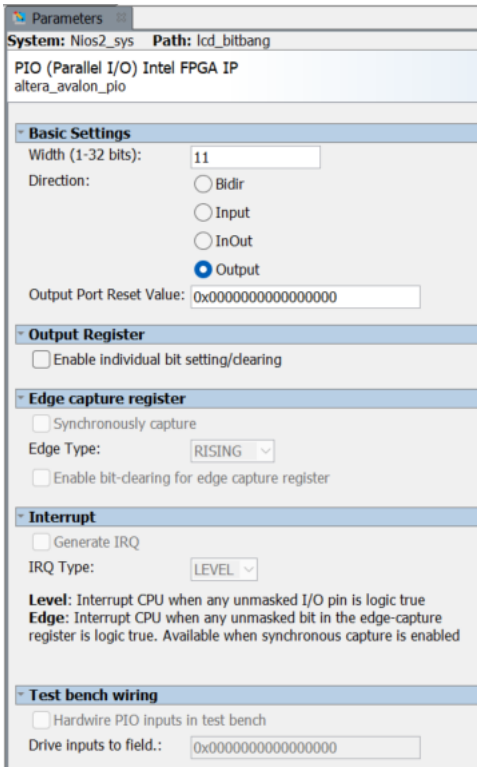


Figure 3-23 LCD PIO setup.

- **LCD Interface Configuration:**

- The PIO's width is set to 11 bits, matching the LCD's data and control signal requirements. This allows for the transmission of the necessary commands and data to the LCD for displaying information.
- The direction of the PIO is configured as output, as it is used to send signals to the LCD.
- The output register and edge capture register are set up to manage the data flow and signal changes, ensuring that the LCD receives clear and stable inputs.
- The interrupt generation feature is enabled, which can be used to alert the Nios II processor when the display is ready for new data or has completed an operation.

By configuring the PIO in this manner, the system can effectively communicate with the LCD display, providing the ability to output real-time information such as traffic sign classifications. This setup ensures that the system can interact with the user, displaying results and statuses directly on the LCD.

3.4.4.1.1.5 Push Button

Figure 3-24 and 3-25 shows our push button PIO block diagram and how the configuration of push button PIO was setup in this project. The push buttons in the Traffic Sign Recognition (TSR) project are integrated into the system using a Parallel I/O (PIO) interface.

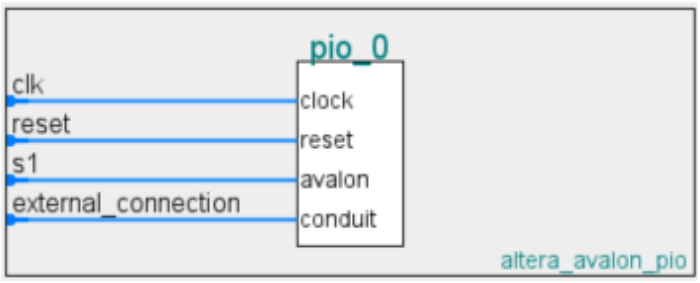


Figure 3-24 Push Button PIO block diagram.

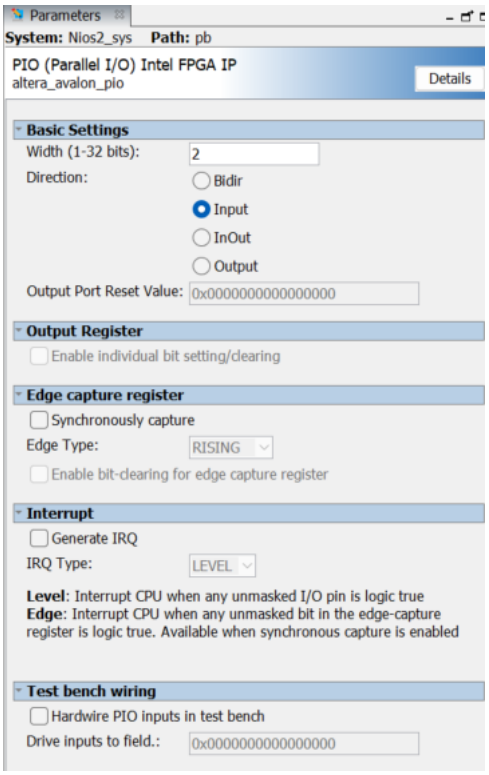


Figure 3-25 Push Button PIO setup.

- **PIO Configuration for Push Buttons:**
  - The PIO's width is set to 2 bits, which corresponds to the number of push buttons used for user input.
  - The direction is configured as input, indicating that the PIO is used to receive signals from the push buttons.
  - The output port reset value is set to zero, ensuring that the button state is clear on reset.

By integrating the push buttons with the appropriate configurations in Quartus Prime, the system is equipped to receive user inputs, which can be used to interact with the Traffic Sign Recognition application, such as triggering the recognition process or cycling through different operational modes.

3.4.4.1.2 Connections

In the hardware design of the Nios II based system for the Traffic Sign Recognition (TSR) project, various components are interconnected to form a cohesive system capable of executing the TSR algorithm. The following Figure 3-23 is showing

the hardware connections, as visualized in the system design connection in Figure 3-12:

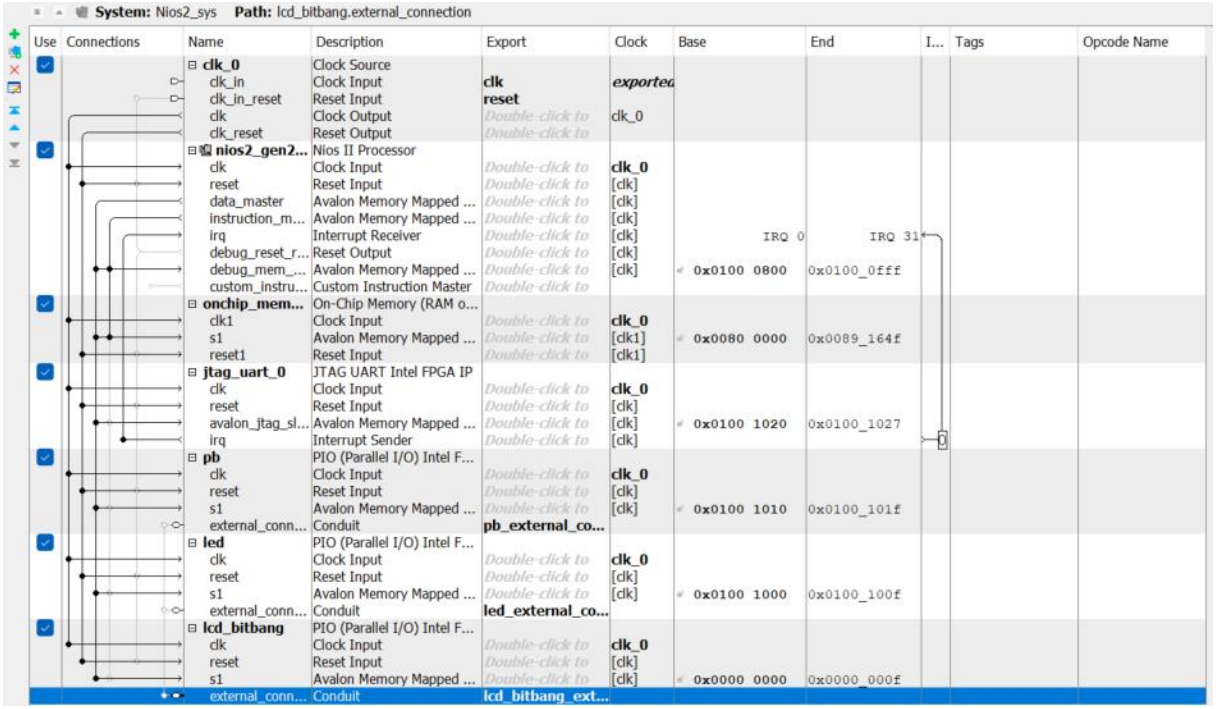


Figure 3-26 Nios II based system connections.

- **Nios II Processor**: The brain of the system, this processor is connected to the Avalon bus, which acts as a highway for data exchange between the processor and the other components in the system.
- **On-Chip Memory**: Directly interfaced with the Nios II processor via the Avalon memory-mapped interface, this memory component provides quick access to data and instruction sets necessary for TSR operations.
- **JTAG UART**: This component is connected to the Avalon bus as a slave, allowing it to communicate with the Nios II processor for debugging purposes and to facilitate data input/output operations through the JTAG interface.
- **Switches**: User inputs via switches are captured through connections to the PIO (Parallel I/O) interface, which is mapped into the system's memory space to allow the Nios II processor to read the switch states as part of its input data stream.
- **LCD Display**: The LCD is driven by a separate PIO block that is configured to send display data from the Nios II processor over the Avalon bus, enabling visual output from the TSR application.

Each component's interface is mapped to a unique address space within the Avalon memory-mapped space, ensuring that data can be routed correctly between the processor and peripherals. Clock and reset signals are distributed across the components to maintain synchronous operation and to ensure that all parts of the system can be reset to a known state when necessary.

The Avalon bus architecture enables modular design and scalability, allowing for the addition of new components as the system requirements evolve. It provides a flexible platform for developing the TSR system, ensuring that the Nios II processor can effectively manage the flow of data for real-time traffic sign recognition.

3.4.4.1.3 Generate the Nios II Based System Design

In this stage of the hardware design phase for the Traffic Sign Recognition (TSR) project, the Quartus Prime software was used to generate the system design information file specifically, a (`.sopcinfo`) file. This file contains important information about the system's components, their connections, and the communication interfaces.

The (`.sopcinfo`) file is a key output of the Platform Designer tool within Quartus Prime. It acts as a blueprint for the Nios II system, detailing the configuration of the processor, the peripherals, and the memory mapping. This file is particularly important because it is used in the software development phase in the Nios II Software Build Tools (SBT) for Eclipse. It enables the SBT to understand the hardware configuration, thus allowing for the creation of software drivers and the application code that will run on the hardware. By generating the (`.sopcinfo`) file, we bridge the gap between hardware and software development, ensuring that the software components are accurately tailored to the hardware they will control and interact with.

3.4.4.1.4 Assign Pin, Location, Timing and Design Constraint

In the Quartus Prime environment, the Pin Planner is a critical tool used to assign specific pins on the FPGA to the various inputs and outputs of our Nios II system design. This step is vital for interfacing the FPGA with external components like the LCD, switches, and JTAG UART. Figure 3-27 shows the GUI Pin Planner to assign pin, location, timing and others design constraints.

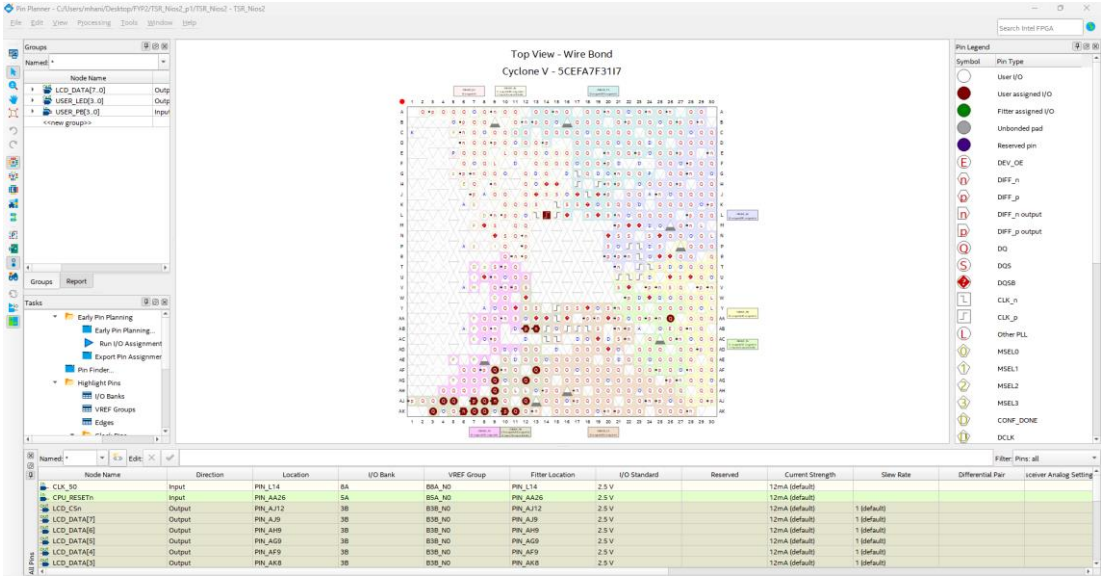


Figure 3-27 GUI Pin Planner to assign pin, location, timing and others design constraints.

Table 2–13. User-Defined Push Button Schematic Signal Names and Functions

Board Reference	Schematic Signal Name	Cyclone V E FPGA Pin Number	I/O Standard
S5	USER_PB0	AB12	2.5-V
S6	USER_PB1	AB13	2.5-V
S7	USER_PB2	AF13	2.5-V
S8	USER_PB3	AG12	2.5-V

Table 2–17. Character LCD Pin Assignments, Schematic Signal Names, and Functions

Board Reference (J14)	Schematic Signal Name	Cyclone V E FPGA Pin Number	I/O Standard	Description
7	LCD_DATA0	AJ7	2.5-V	LCD data bus
8	LCD_DATA1	AK7	2.5-V	LCD data bus
9	LCD_DATA2	AJ8	2.5-V	LCD data bus
10	LCD_DATA3	AK8	2.5-V	LCD data bus
11	LCD_DATA4	AF9	2.5-V	LCD data bus
12	LCD_DATA5	AG9	2.5-V	LCD data bus
13	LCD_DATA6	AH9	2.5-V	LCD data bus
14	LCD_DATA7	AJ9	2.5-V	LCD data bus

Table 2–17. Character LCD Pin Assignments, Schematic Signal Names, and Functions

Board Reference (J14)	Schematic Signal Name	Cyclone V E FPGA Pin Number	I/O Standard	Description
4	LCD_D_Cn	AK11	2.5-V	LCD data or command select
5	LCD_WEn	AK10	2.5-V	LCD write enable
6	LCD_CSn	AJ12	2.5-V	LCD chip select

Table 2–10. On-Board Oscillators

Source	Schematic Signal Name	Frequency	I/O Standard	Cyclone V E FPGA Pin Number	Application
U4	CLKIN_50_FPGA_TOP	50.000 MHz	Single-Ended	L14	Top and right edge
	CLKIN_50_FPGA_RIGHT			P22	
X3	CLK_CONFIG	100.000 MHz	2.5V CMOS	—	Fast FPGA configuration
X1 and U3 (buffer)	DIFF_CLKIN_TOP_125_P	125.000 MHz	LVDS	L15	Top and bottom edge
	DIFF_CLKIN_TOP_125_N			K15	
	DIFF_CLKIN_BOT_125_P			AB17	
	DIFF_CLKIN_BOT_125_N			AB18	

Figure 3-28 The table show the list of user I/O interface and voltages required for the Cyclone V E Development Kit FPGA, including the push buttons, LCD, and on-board oscillator (for clock).

- Assigning each signal from our Nios II system to a physical pin on the FPGA, ensuring that each connection corresponds with the physical layout of the board and the I/O requirements.

- Setting the location constraints, which involves specifying the exact pins on the FPGA that each signal should be routed to. This is done by cross-referencing the FPGA's pinout with the design's I/O needs.
- Configuring the I/O standards for each pin, which must match the voltage levels and signal characteristics of the external devices they connect to. In the Figure 3-25, we see various pins set to a 2.5V I/O standard, suitable for many CMOS-based interfaces.
- Defining the timing constraints, which are crucial for the system to operate reliably at the desired clock frequencies. This includes setting up the input and output delays and ensuring the design meets the timing requirements during the compilation process.
- Adjusting other design constraints such as current strength and slew rate, which affect how quickly and powerfully the signal is driven onto the pin. These settings can impact signal integrity and must be tuned to match the electrical characteristics of the connected hardware.

After completing these assignments, the design must pass through Quartus Prime's compilation process, which includes synthesis, fitting, and timing analysis. This ensures that the pin assignments are physically realizable on the FPGA and that the design will function correctly with the connected hardware components.

3.4.4.1.5 Compile Hardware for Target Device

Once the pin assignments and constraints have been set up in the Quartus Prime software, the next step is to compile the hardware for the target FPGA device. This is a multi-stage process that includes:

1. **Synthesis**: The high-level design is transformed into a gate-level representation. The synthesis tool interprets the HDL (Hardware Description Language) code and generates a netlist.
2. **Fitting (Place and Route)**: The synthesized netlist is then fitted onto the FPGA's physical architecture. The fitting process involves placing the logic into specific locations on the FPGA and routing the connections between them.

3. **Timing Analysis**: After fitting, the design undergoes timing analysis to ensure that all timing constraints are met. This is crucial for the system's reliability and performance, as it ensures that signals propagate through the logic in the FPGA within the required timeframes.
4. **Bitstream Generation**: Upon successful timing analysis and fitting, Quartus Prime generates the .sof file, which contains the configuration data for the FPGA. This SRAM Object File is the final output of the compilation process and is used to program the FPGA.
5. **Programming the Device**: The .sof file is downloaded to the FPGA using a downloader cable (like USB-Blaster) or other programming interfaces. This configures the FPGA's volatile memory, effectively "programming" the FPGA with the designed hardware system.

By the end of this process, the Cyclone V E FPGA is equipped with all the logic and functionality described in the Nios II system design and is ready for real-world deployment in the Traffic Sign Recognition application.

3.4.4.2 Software Design

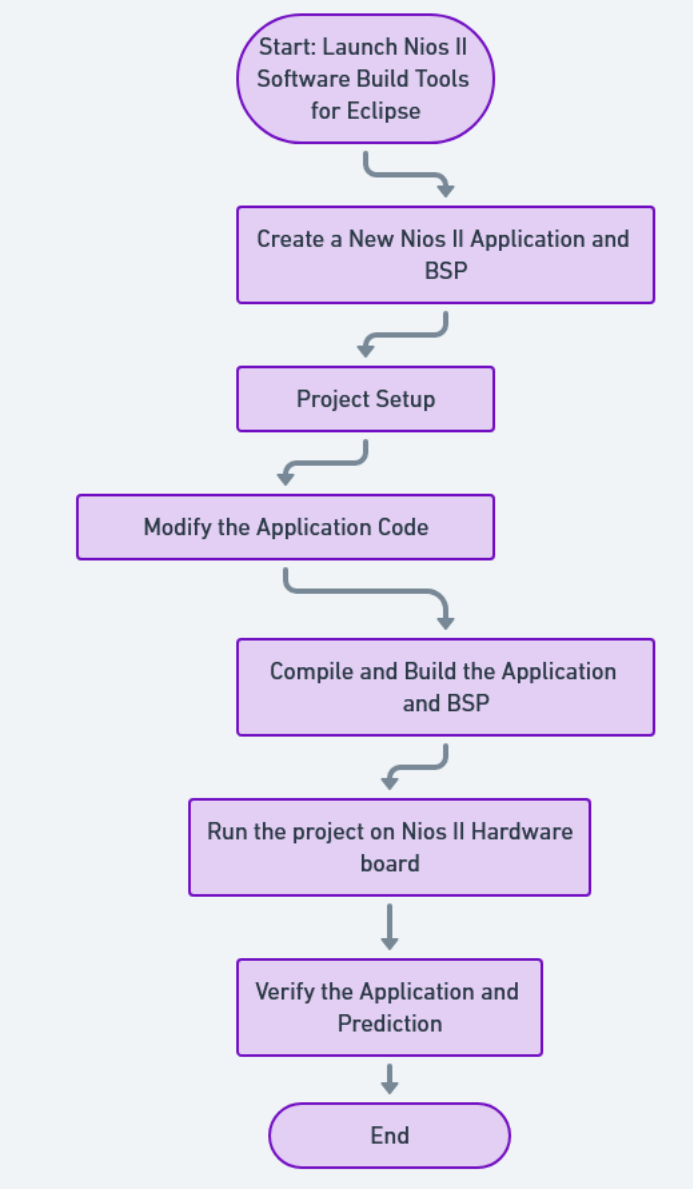


Figure 3-29 Software Design Flow for Traffic Sign Recognition.

Figure 3 29 illustrates the software design flow for Traffic Sign Recognition. The software design for the Traffic Sign Recognition (TSR) system was meticulously architected, beginning with the launch of the Nios II Software Build Tools for Eclipse, a new Nios II project application, coupled with the Board Support Package (BSP), to create and formed the foundation of the system. The project setup was to place the main source code and the others source code or header file in the project application. Modifications of the application code were influenced by the algorithm's specific

needs, like to ensure that the neural network model received and processed image data correctly, which involved normalizing and structuring the input to align with the network's requirements or setting the user interface for the TSR application.

The compilation and building of the application and BSP transformed the high-level code into a machine-readable format, facilitating the execution on the Nios II hardware board. This deployment was a critical step towards application and testing, where the software's performance could be evaluated in an embedded system context. Running the project on the hardware board was followed by a process of verification, ensuring that the predictions made by the neural network were accurate and reliable. The display of results was handled through functions designed to interface seamlessly with peripheral devices, displaying the classification outcomes. This holistic approach to software design, encompassing development, deployment, and validation, ensured the efficacy of the TSR system on the Intel Cyclone V E FPGA.

3.4.4.2.1 Starting with Nios II Software Build Tools for Eclipse

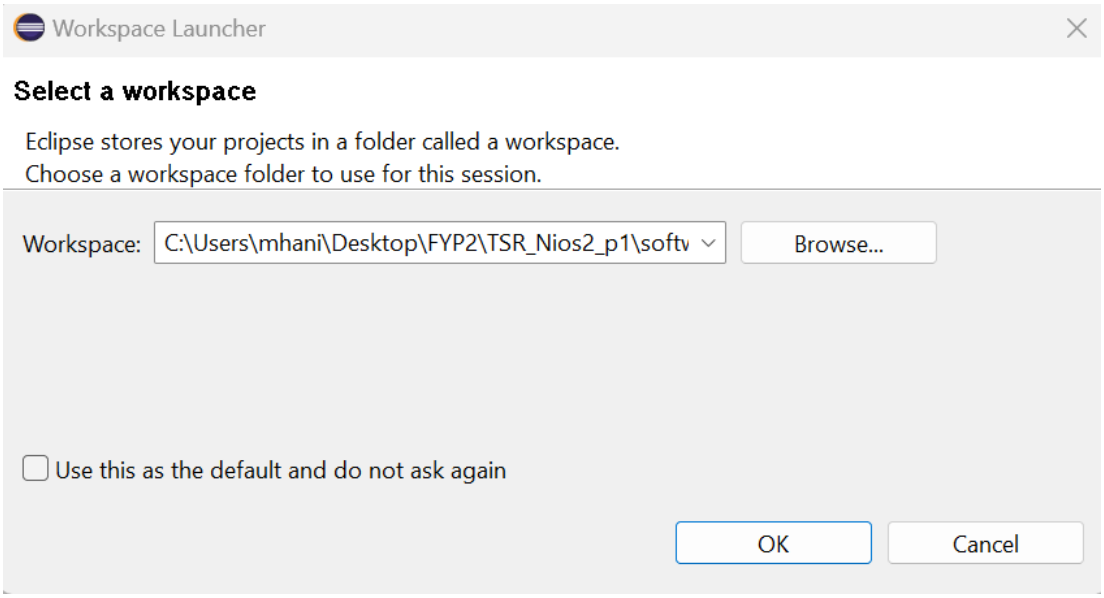


Figure 3-30 Open workspace for Nios SBT for Eclipse.

When we launch Nios II Software Build Tools for Eclipse, establishing the workspace directory within the 'software' folder of the Quartus project directory is a strategic decision in the development process. Figure 3-30 show how we establishing the workspace directory within the 'software' folder of the Quartus project directory. This alignment ensures that all software-related files and scripts remain organized and

are inherently linked to the corresponding hardware project. It facilitates a streamlined workflow, as the Eclipse environment and Quartus Prime can share resources and references without conflict. Moreover, this structure simplifies version control and backup procedures, as the software and hardware configurations are contained within a unified project directory, thereby enhancing the maintainability and scalability of the project.

3.4.4.2.2 Creating the Nios II Application and BSP and Project Setup

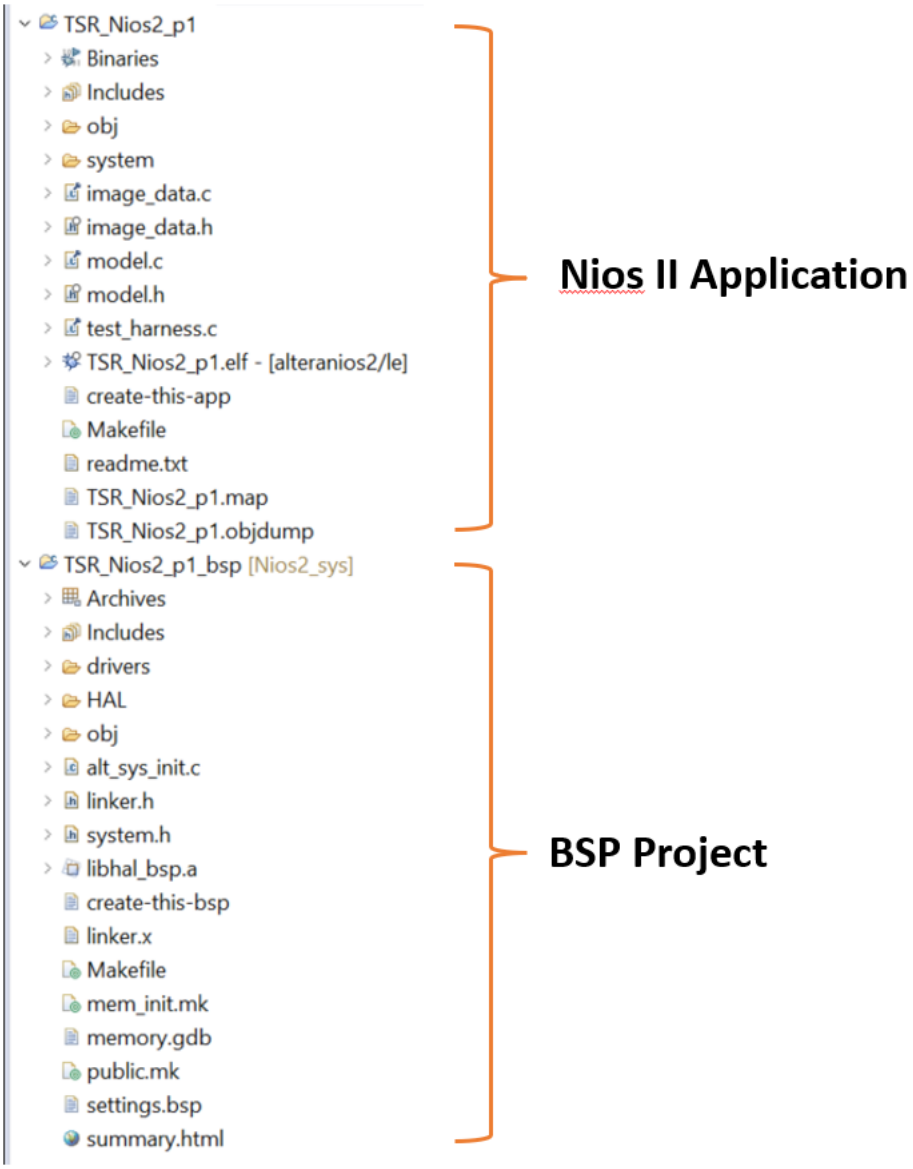


Figure 3-31 Creating Nios II Application and BSP.

In the creation of the Nios II Application and Board Support Package (BSP), the synergy between the two components is paramount for the successful deployment

of the Traffic Sign Recognition system. The Nios II application houses the core logic of the TSR algorithm, encapsulating the operational code that defines the behavior of the system. The BSP, on the other hand, is essential as it contains all the drivers and API necessary for the application to interface with the hardware effectively. Figure 3-31 shows the creation of the separate application and BSP project within the Nios II SBT for eclipse platform.

Creating the project for the Nios II application and BSP begins with referencing the `'.sopcinfo'` file from the hardware design, which provides information about the system's hardware configuration to the software environment. This file ensures that the software is tailored to the specific hardware components and resources, facilitating seamless communication between the application and the underlying hardware. The BSP is automatically generated to match the hardware described in the `'.sopcinfo'` file, which is critical as it abstracts the hardware complexities, allowing the application developers to focus on implementing the high-level TSR logic without worrying about the lower-level hardware interactions.

In the BSP project structure, the `system.h` file typically contains macros, declarations, and definitions that are derived from the system's hardware configuration. It acts as the interface between the hardware and the software, providing a layer of abstraction that allows the software to interact with the hardware components without needing to manage the low-level details. This might include definitions for system clock settings, peripheral addresses, and interrupt controllers.

The `linker.h` (or similarly named files such as `linker.ld` or `linker.script`) is concerned with the memory layout of the application. It defines where in memory the different sections of the code and data will reside, such as the heap, stack, text, data, and bss segments. This file ensures that the compiled code is correctly mapped to the physical memory of the device, which is for the proper execution of the software on the hardware.

The relationship between the application and BSP is akin to that of a brain and its neural network; the application represents the decision-making center, while the BSP provides the pathways to interact with the body, which in this context is the Nios II based system hardware. This cohesive relationship is critical as it ensures that the

application can effectively utilize the hardware to achieve optimal performance in real-time traffic sign recognition.

3.4.4.2.3 Program Setup

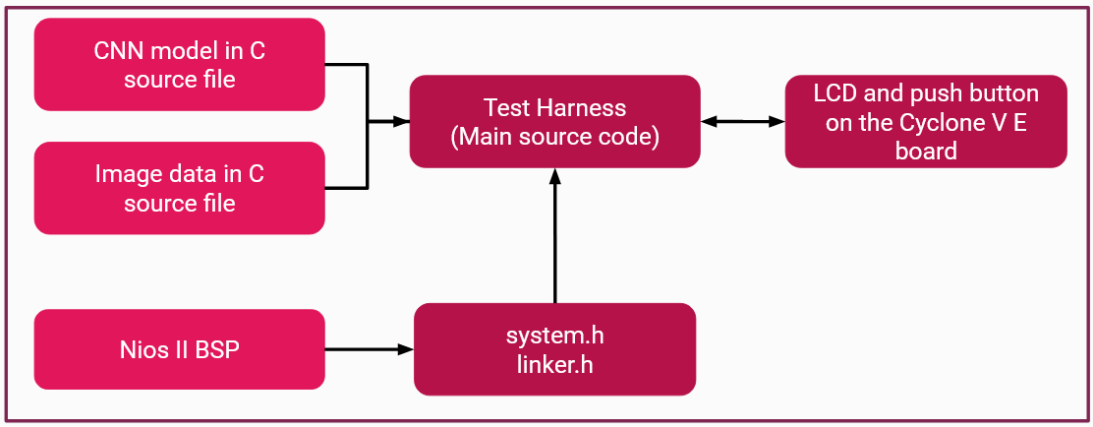


Figure 3-32 Traffic Sign Recognition program setup.

Figure 3 32 illustrates the traffic sign recognition program setup. The main source code is central to the application's operation. This C source file is configured to interact with the hardware-defined in the Quartus project through the inclusion of `'system.h'`, which provides necessary hardware abstractions and image data stored for the traffic sign data. The setup involves configuring the Eclipse environment to work with these files, ensuring the compiler options are set to optimize for the Nios II processor's characteristics and the application's memory constraints. It incorporates the Convolutional Neural Network (CNN) model from `'model.c'`, which contains the logic for image recognition. The image data, essential for the CNN's operation, is stored in a separate C source file, likely `'image_data.c'`. Both the CNN and image data are interfaced with the main source code, ensuring a cohesive workflow. The Nios II Board Support Package (BSP) provides the necessary system configurations and drivers, as referenced by `'system.h'`, and the memory allocation specifics managed by `'linker.h'`. The outcome of this setup is the ability of the software to drive outputs, such as displaying results on an LCD and receiving inputs from push buttons on the Cyclone V E board, thus providing an interactive user experience.

3.4.4.2.4 Modifying the Application Code

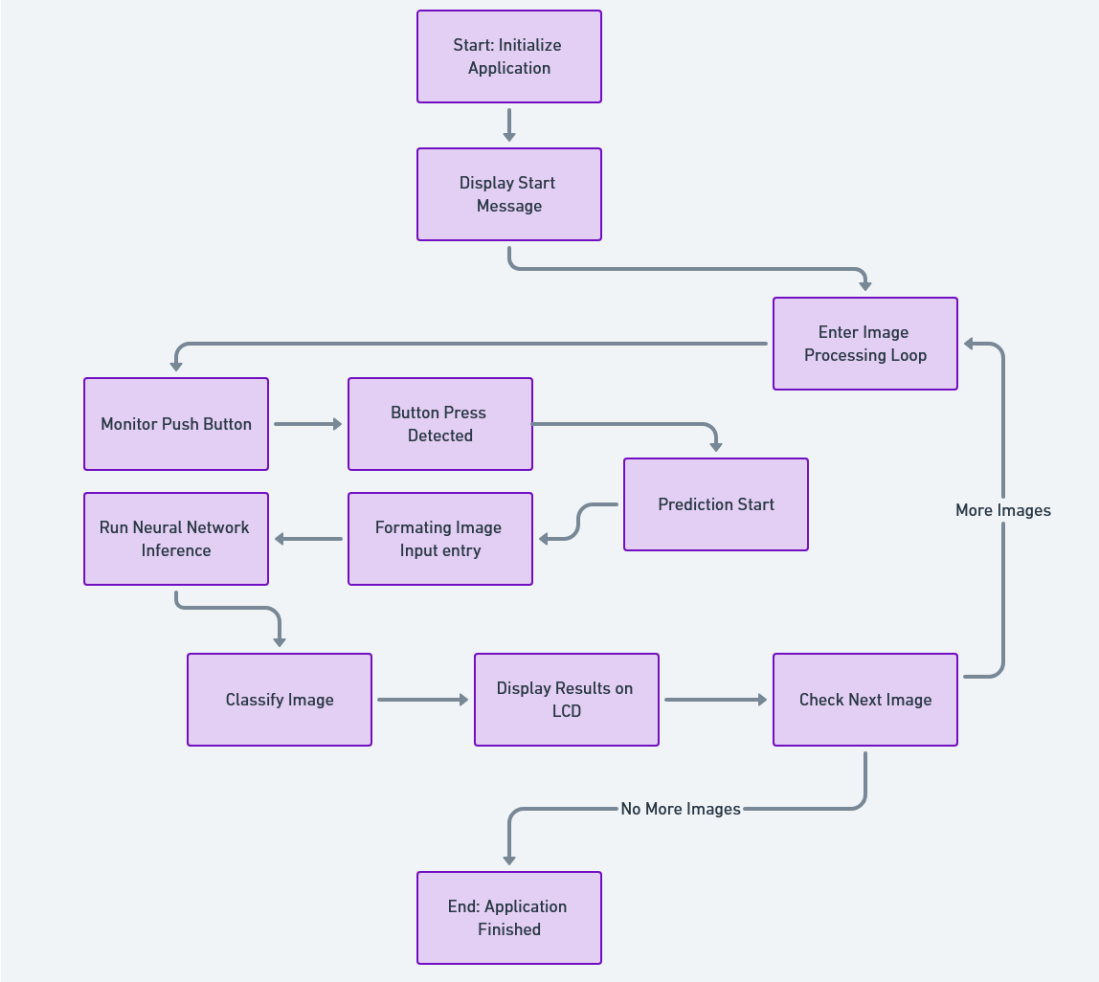


Figure 3-33 Main application of our TSR system flow process.

The flowchart in Figure 3-30 illustrates the step-by-step execution of the Traffic Sign Recognition application with hardware interaction.

1. **Initialization**: The program starts with an initialization sequence, displaying a greeting message on the LCD screen, indicating the application has begun.

2. **Monitoring Button Press**: The application enters a loop that constantly monitors the push button status. Once the button is pressed, the loop breaks, and the program proceeds to process an image.

3. **Image Formatting**: Before feeding into the CNN, the image data is formatted from the raw image array to a normalized form that the neural network can process.

4. **Neural Network Inference**: The formatted image is then passed to the CNN, which runs an inference to classify the image. The CNN processes the image and produces a tensor representing the classification results.

5. **Classification and Output**: The program determines the classification with the highest confidence score and displays the corresponding traffic sign message on the LCD.

6. **Loop or Terminate**: If there are more images to process, the program loops back to monitor for another button press; otherwise, it terminates with a completion message.

Throughout the process, the program uses utility functions for delays and to interface with the LCD through command and data registers, managing the display of messages and results.

3.4.4.2.4.1 Main Application Code Structured

The code begins by including necessary headers, such as `stdio.h` for standard input/output operations, and specific headers like `system.h` and `altera_avalon_pio_regs.h` that are essential for interfacing with the FPGA hardware. Notably, `image_data.h` is included to access the image data required for sign recognition, and `model.h` encapsulates the neural network model that performs the classification.

```
#include <stdio.h>
#include "sys/alt_stdio.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include "io.h"
#include "model.h" // Ensure model.c contains the entry function
#include "image_data.h" // Contains the raw_image_data array
```

Following these inclusions, we define constants for image size and declare an external array, `raw_image_data`, which is populated with the processed image data. A buffer, `output_tensor`, is also declared to store the output from the neural network.

```
#define IMAGE_SIZE (32 * 32 * 3) // Size for a 32x32 RGB image
extern const uint8_t raw_image_data[NUMBER_OF_IMAGES][IMAGE_SIZE]; // Defined in image_data.c
float output_tensor[1][4]; // Buffer for the output from the neural network
```

The `main` function marks the entry point of the application. It initiates with a welcome message and immediately calls `display_start`, a function that primes the LCD display with a greeting, signifying the system's readiness.

```
void display_start(){
    alt_putstr("Hello from Nios II!\n");

        // Initialize lcd
        SendCommand(0x0038);  // Function set: 8 bit, 2 lines 5*8 dots
        SendCommand(0x000F);  // Display on, cursor on, cursor blinking
        SendCommand(0x0001);  // Display clear
        SendCommand(0x0006);  // Entry mode: right-moving cursor (address increment), no display shift

        // Write first line message to lcd
        SendMessage("Traffic Sign");

        // Change DDRAM location to 40H to map to the second line
        SendCommand(0x00C0);  // Set DDRAM address to 40H

        // Write second line message to lcd
        SendMessage("Recognition");
}

int main() {
    alt_putstr("Starting the application...\n");
    display_start();
```

As we journey through the `main` loop, the system awaits user interaction, signified by a push button press. Upon activation, it retrieves an image from `raw_image_data`, formats it for the neural network, and calls `entry`, a function from `model.h`, which feeds the image into the neural network for classification.

```
for (int img_idx = 0; img_idx < NUMBER_OF_IMAGES; ++img_idx) {
    // Wait for push button to be pressed
    alt_putstr("Monitoring push button...\n");
    while (1) {
                int button_data = IORD_ALTERA_AVALON_PIO_DATA(PB_BASE) & 0x3; // Assuming you are using the first button
                if (button_data == 1) { // Check if the first button is pressed
                    alt_putstr("button pressed...\n");
                    break; // Exit the waiting loop
                }
            }
    // Debounce delay to avoid multiple reads from a single press
    simple_delay(100000); // Adjust delay as needed for your system
    float formatted_input[1][32][32][3]; // Array to hold formatted input
    alt_putstr("Count loop 1...\n");
    for (int i = 0; i < IMAGE_SIZE; ++i) {
        formatted_input[0][i / 96][(i % 96) / 3][i % 3] = raw_image_data[img_idx][i] / 255.0f;
    }

    float output_tensor[1][4]; // Buffer for the output from the neural network
    entry(formatted_input, output_tensor); // Use formatted input for inference

    int predicted_class = 0;
    float max_value = output_tensor[0][0];
    for (int i = 1; i < 4; ++i) {
        if (output_tensor[0][i] > max_value) {
            max_value = output_tensor[0][i];
            predicted_class = i;

        }
    }
```

```
        display_results(predicted_class);
        alt_putstr("Classification completed.\n");
        char predicted_class_str[12];
        sprintf(predicted_class_str, "%d", predicted_class);
        alt_putstr("Predicted Class: ");
        alt_putstr(predicted_class_str);
        alt_putstr("\n");


    }

    alt_putstr("Application finished.\n");
    return 0;
}
```

The prediction is then passed to `display_results`, a function that converts the neural network's numerical output into a human-readable format and presents it on the LCD display. The messages corresponding to the classification results, such as "No U-turn" or "Speed Limit 30", are not arbitrary strings but are critical pieces of information for the driver or autonomous vehicle system.

```
void display_results(int classification) {
    // Set up the display or send commands to another output as needed
    // Here, it's set up to use an LCD display connected to the system
    alt_putstr("Displaying result on LCD...\n"); // Debug message

    // Initialize the LCD display
    SendCommand(0x0038); // Function set: 8-bit, 2 line, 5x8 dots
    SendCommand(0x000F); // Display on, cursor blinking
    SendCommand(0x0001); // Clear display
    SendCommand(0x0006); // Entry mode: cursor moves to the right
    // Write first line message to lcd
    SendMessage("Predicted as:");

    // Change DDRAM location to 40H to map to the second line
    SendCommand(0x00C0);  // Set DDRAM address to 40H

    // Array of messages corresponding to the classification indices
    char* messages[] = {
        "No U-turn",
        "Speed Limit 30",
        "Stop",
        "Yield",
        "Unknown Sign"
    };

    // Ensure the classification index is within bounds before accessing the array
    char* message = (classification >= 0 && classification < 4) ? messages[classification] : messages[4];

    // Send the message to the LCD
    SendMessage(message);
}
```

The code is punctuated by debugging messages, which serve as signposts, guiding the developer through the execution flow and aiding in troubleshooting. Additionally, functions like `simple_delay` manage timing, while `SendCommand` and `SendData` handle the communication with the LCD through a bit-banging protocol, showcasing the low-level control over the FPGA's I/O pins. In a more granular sense, `SendCommand` and `SendData` embody the hardware interaction as they directly manipulate the LCD's command and data registers. These functions utilize the I/O write operations (`IOWR_ALTERA_AVALON_PIO_DATA`) provided by

the system-level interface to send the appropriate signals to the LCD. The bit-banging approach used here is a testament to the flexibility of FPGA-based systems, allowing developers to implement custom communication protocols that aren't natively supported by the hardware. The `SendMessage` function iterates through a string, character by character, displaying the message on the LCD. This is crucial for providing feedback to the user and for debugging purposes.

```
void simple_delay(unsigned int delay)
{
    for (unsigned int i = 0; i < delay; i++) {
        // This is a simple busy-wait loop.
        __asm__ volatile ("nop");
    }
}

void SendCommand(alt_u8 cmd) // bitbang
{
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0400 | cmd);
    simple_delay(10000); // Adjust the delay value as needed
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0000 | cmd); // Enable
    simple_delay(10000);
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0400 | cmd);
    simple_delay(10000);
}

void SendData(alt_u8 data) // bitbang
{
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0600 | data);
    simple_delay(10000);
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0200 | data); // Enable
    simple_delay(10000);
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0600 | data);
    simple_delay(10000);
}


void SendMessage(char *msg)
{
    for (; *msg != 0; msg++)
    {
        SendData(*msg);
    }
}
```

The provided code encapsulates a series of complex interactions between the FPGA's soft-core Nios II processor and the peripheral LCD display. It demonstrates how raw image data is transformed into a comprehensible visual output, illustrating the model's predictions in a format that is readily understandable. The application code is more than just a series of C instructions; it is the bridge between the abstract world of software and the tangible realm of hardware. It showcases the practical application of theoretical knowledge and complex algorithms in a real-world system.

3.4.4.2.5 Compiling and Building the Application and BSP

After finishing the modification of the application code, then we can proceed to compile and build the project. In this phase, the application and BSP undergo a compilation process, translating and integrating the high-level instructions into a machine-readable format suitable for the embedded environment. This will generate the `(.elf)` file which a machine-readable format for Cyclone V E board.

3.4.4.2.6 Running the Project on Nios II Hardware Board

The compiled application is deployed and executed on the Nios II hardware, marking the software-hardware interaction.

3.4.4.2.7 Verifying the Application and Prediction

The final stage involves a thorough verification of the software's functionality and the TSR system's predictive accuracy, ensuring the system's readiness for deployment.

3.5 Summary

The methodology for implementing Traffic Sign Recognition (TSR) on the Intel FPGA Cyclone V E Development Kit involves a series of systematic steps. Initially, data is collected, including images of Malaysian road signs, which are then used to train a Convolutional Neural Network (CNN) model. The training process involves feeding the images to CNN to enable it to learn and accurately identify various road signs. Once the model is trained and tested for accuracy, the next challenge is to convert the model, originally developed in Keras, into C source code. This conversion is crucial for the implementation of the TSR system on the Nios II soft core processor within the FPGA. The final step is the actual implementation of the model on the FPGA platform, where it is tested in a real-time environment to recognize and interpret traffic signs effectively. This methodology is integral to achieving the thesis objective of developing a functional and efficient TSR system using FPGA technology, particularly for the recognition of Malaysian road signs in an embedded system context.

CHAPTER 4

RESULT & DISCUSSION

In this chapter, we show the procedure in demonstrating our project from inserting the input and getting the output. It also contained our result analysis based on the result we obtained and discussion on what are the limitations in this project and the reason why is that occurred.

4.1 TSR Demonstaration

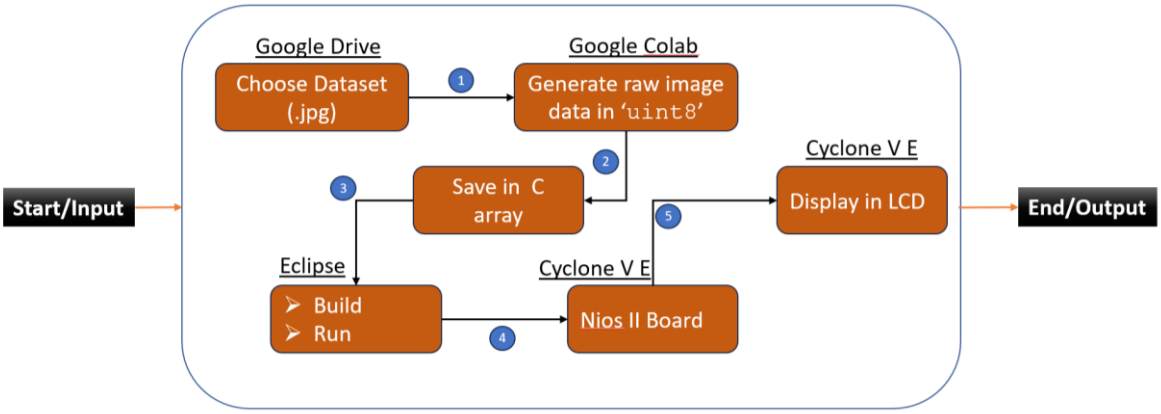


Figure 4-1 Project setup for demonstration.

Figure 4-1 illustrates how the project demonstration need to setup. The demonstration begins with dataset selection from Google Drive, outlining how raw image data is processed in Google Colab. The data is then converted into a C array format, enabling integration with the Cyclone V E development environment. The culmination of the process is executed on the Nios II board, where the built and run actions take place in the Eclipse environment. Finally, the results are displayed on an LCD screen, showcasing the system's ability to accurately detect and classify traffic signs.

4.1.1 Data Selection

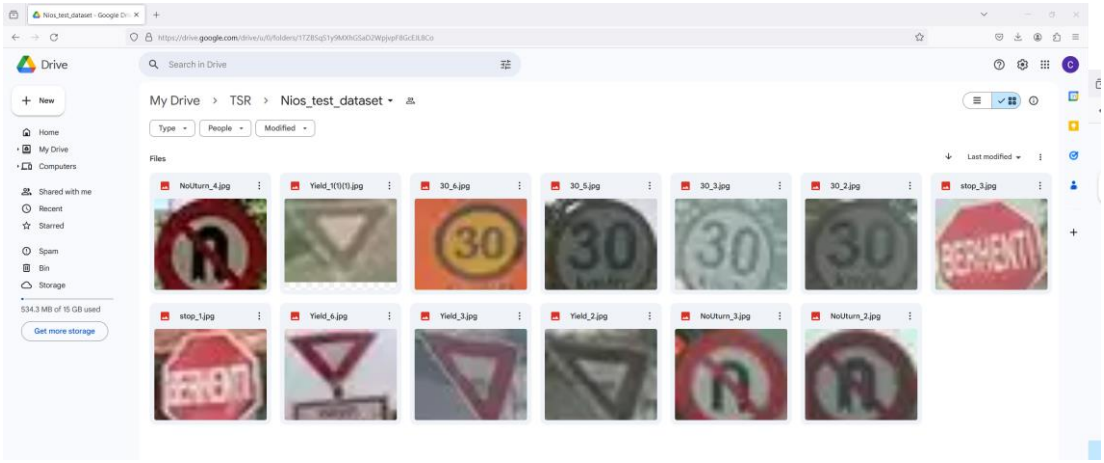


Figure 4-2 Upload the chosen image dataset to be predict.

The process begins with the selection of the dataset and upload it to Google Drive. This dataset consists of .jpg image files of various Malaysian traffic signs that the system will be trained to recognize. Figure 4-2 shows how the chosen test dataset has been upload to Google Drive to be predict.

4.1.2 Image Data Generation

In the development of the Traffic Sign Recognition (TSR) system, a pivotal step involved the transformation of visual data into a format interpretable by the system's machine learning algorithm. This process, termed 'Image Data Generation', necessitated the conversion of image files into numerical arrays that could be handled by the C programming language within the embedded environment.

Utilizing the Python programming language, a script (can refer in Appendix ()) was crafted employing two key libraries: Pillow for image processing and NumPy for numerical operations. The script was executed as follows:

1. **Dataset Retrieval**: Initiated by specifying the directory path on Google Drive, the script accessed a collection of .jpg images, each depicting a Malaysian traffic sign.
2. **Output File Initialization**: An output file path was declared, designating the location for the processed image data to be stored. This data would later be assimilated into the system's C source code.

3. **Image File Collection**: The script employed the `glob` module to compile a list of image files from the designated folder path.
4. **Image Resizing**: The images were standardized to a resolution of 32x32 pixels, a necessary step to maintain uniformity in data input. This resolution resulted in an array of 3,072 values for each image, accounting for the RGB channels.
5. **Array Declaration**: An output file was initiated to store the image data. The script began by writing the declaration of a multi-dimensional array in C syntax.
6. **Image Processing**: Each image underwent a series of processing steps:
    - **Loading and Resizing**: Images were loaded and resized to the predetermined dimensions.
    - **Flattening**: The images were converted into a one-dimensional array of RGB values, flattening the pixel data.
    - **Data Formatting**: These values were then formatted into a string of comma-separated values.
    - **Array Population**: Each image's data string was appended as a row in the C array.
7. **Array Completion**: Post-processing of all images culminated in the closure of the array within the output file.
8. **Confirmation of Completion**: A message was displayed, confirming the successful storage of the image data.

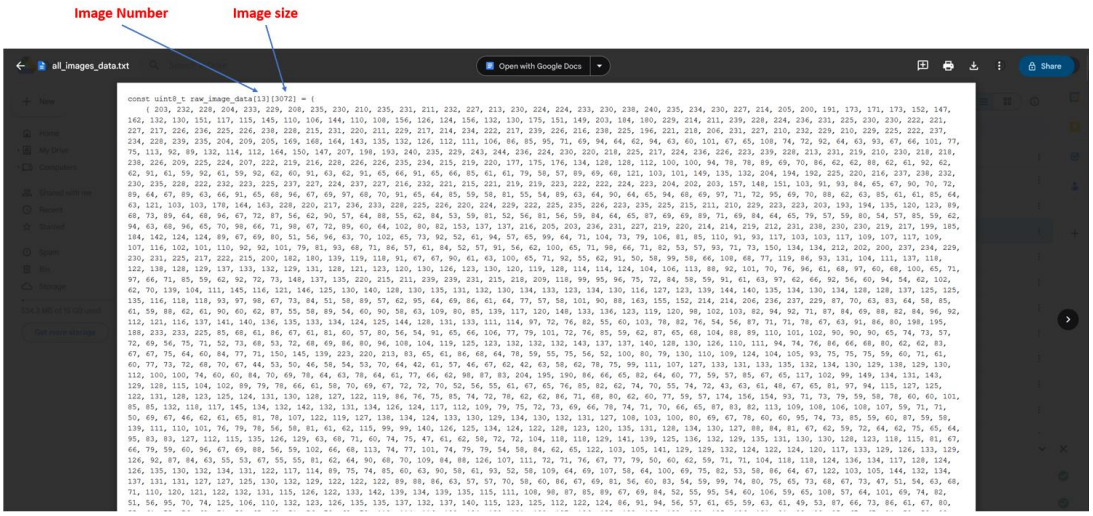


Figure 4-3 The output of image data that has been generate in 'uint8' type.

Figure 4 3 show the output of image data that has been generate in 'uint8' type. The output, as captured in the image provided, presented a text file populated with rows of numerical values. Each row corresponded to the raw data of an image, systematically arranged to represent the pixel values. These arrays formed the crux of the TSR system's input data, allowing for the evaluation of the recognition algorithm's performance on the Nios II board.

4.1.3 Storing Image Data

Figure 4-4 Our filename for image data bank in this project.

The process of data storage within the context of a Traffic Sign Recognition (TSR) system is a meticulous one, as it necessitates the image data to be saved in a format that is both accessible and usable by the main application source code. The image data generated and stored as a C array must be correctly interfaced with the rest of the embedded system to facilitate the classification tasks performed by the Convolutional Neural Network (CNN) model. This interfacing is achieved through the use of a header file, which acts as a bridge between the raw image data and the application code that processes it.

In this project, the header file, named as `image_data.h` (can refer in Appendix C), serves several purposes:

1. **Declaration of Constants**: It defines constants such as `IMAGE_SIZE`, which represents the size of each image in the array, and `NUMBER_OF_IMAGES`, which indicates the total number of images stored in the array.
2. **Inclusion of Standard Integer Library**: By including `<stdint.h>`, the header file ensures that the `uint8_t` data type, representing an 8-bit unsigned integer, is available. This data type is used to store the pixel values in the range of 0-255, reflecting the standard 8-bit depth of RGB images.
3. **External Declaration**: The `extern` keyword is used to declare `raw_image_data`, indicating that while the data is defined elsewhere, it is

accessible from other source files that include this header. This is a key aspect of modular programming in C, allowing for separation between data definitions and declarations.

4. **Guardians**: The `#ifndef` and `#endif` preprocessor directives ensure that the header file's content is only included once, preventing multiple inclusions that could lead to errors during compilation.

In practice, the `image_data.c` source file would contain the actual array definition and the initialized data that the header file declares as `extern`. When the main application source code requires access to this image data for processing by the CNN model, it includes `image_data.h`. The compiler links the external declaration in the header file with the actual data defined in the `image_data.c` file.

This setup ensures that the image data is neatly encapsulated and can be efficiently called upon by the classification code in the main application, thus streamlining the process of image classification within the TSR system. It exemplifies a fundamental practice in embedded systems programming, where data and code are meticulously organized to facilitate clarity, maintainability, and performance.

4.1.4 Code Compilation and Execution

Within the Eclipse Integrated Development Environment (IDE), the code that constitutes the TSR system is compiled and executed. This involves the 'Build' process, which translates the high-level C code into a lower-level machine code which `'.elf'`, and the 'Run' process, which executes the compiled program. Figure 4-5, 4-6, and 4-7 show the step to verify the code compilation was successfully executed and ready to run.

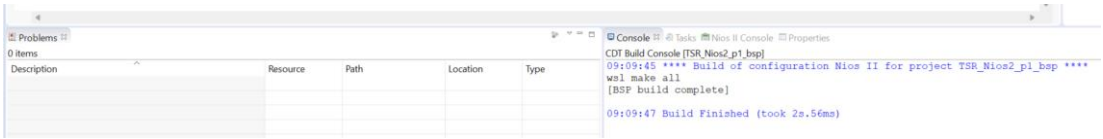


Figure 4-5 Successfully compile and build the project without any errors.

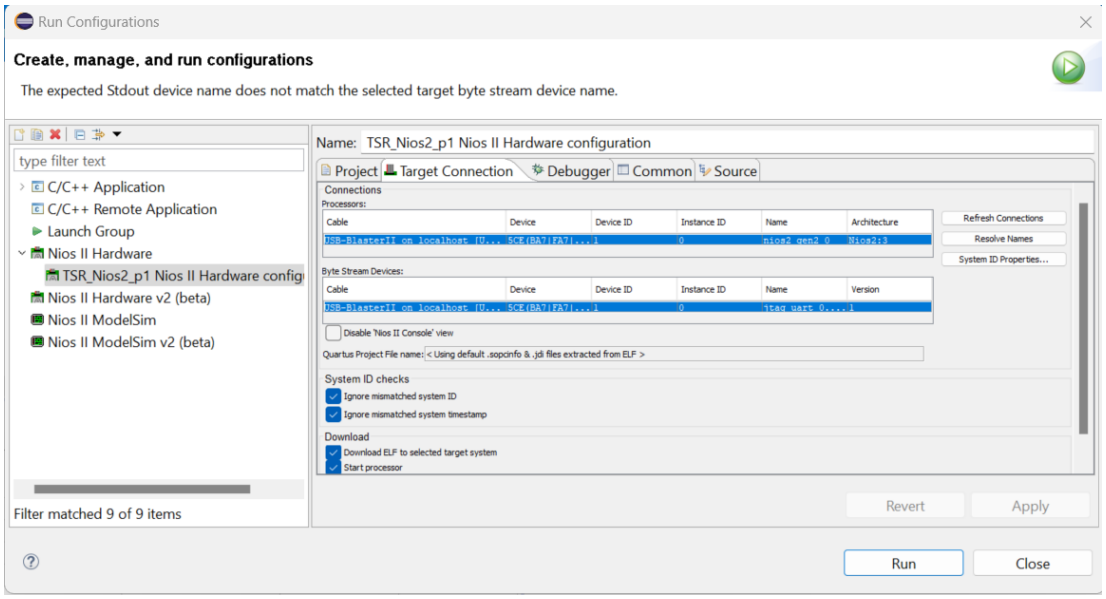


Figure 4-6 Set the target hardware to upload the TSR application.

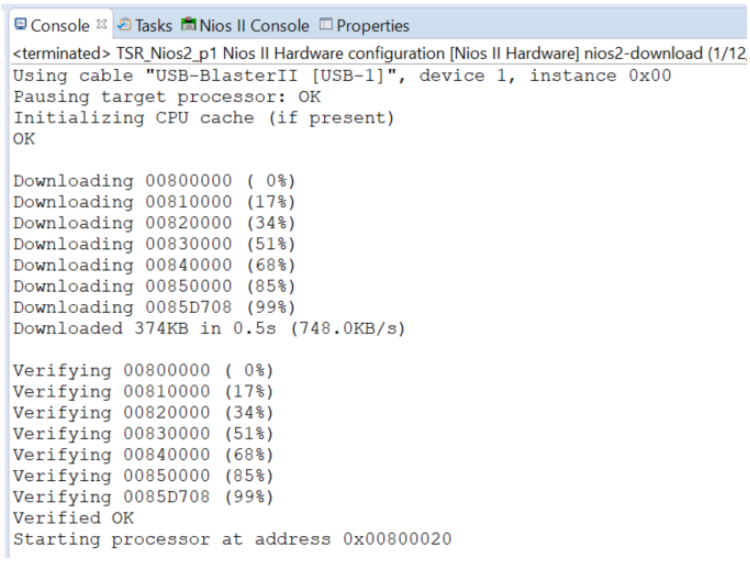


Figure 4-7 The console output of the Nios II IDE shows the hardware configuration file has been successfully downloaded to the target device and ready to run.

4.1.5 Hardware Interaction

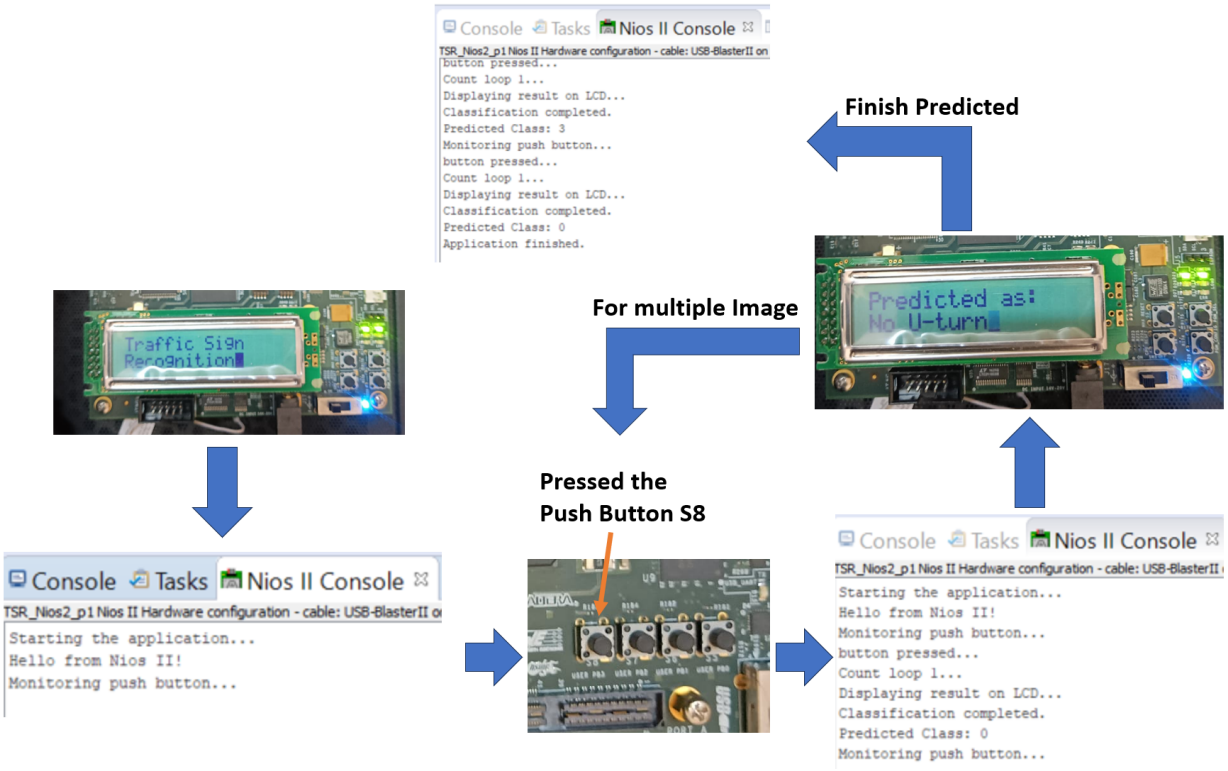


Figure 4-8 Hardware interaction with our TSR system.

Figure 4-8 shows how was the hardware interaction with our TSR system. The Traffic Sign Recognition (TSR) system demonstrates a seamless hardware interaction facilitated through a carefully orchestrated sequence of operations. The process is initiated as soon as the system is powered on, as evidenced by the following steps:

1. **Initialization**: The moment the system is activated, the LCD screen comes to life, prominently displaying the message "Traffic Sign Recognition." This initial greeting serves as an intuitive indicator that the system is live and awaiting user input.
2. **Monitoring Button Press**: In a state of anticipation, the system diligently monitors the push button S8. The embedded software continuously checks the status of this button, effectively pausing the workflow until the user's action is registered. This monitoring phase is a critical user interface element, allowing the application to respond to real-time user commands.
3. **Button Interaction**: The interaction is captured instantaneously when the user presses the button, the system acknowledges this input with a prompt

acknowledgment in the console, displaying the message "button pressed..." This detection marks the transition from a waiting state to active processing.

4. **Image Processing and Formatting**: Following the button press, the system retrieves the next image from its memory banks. This image, initially stored in a raw array format, undergoes a transformation process, normalizing the data to conform to the neural network's input requirements. It is a crucial step that adapts the raw pixel values into a standardized form the CNN can interpret.
5. **Neural Network Inference**: With the image data now in its appropriate format, it is funnelled into the core of the system the CNN model. This model, acting as the analytical brain of the operation, processes the image and computes a set of outputs, known as a tensor, which encapsulates the likelihood of the image representing each of the known traffic signs.
6. **Classification and Output**: Upon processing the data, the system determines the most probable classification based on the confidence scores encoded within the tensor. The classification with the highest score is then selected, and the corresponding traffic sign message, such as "Predicted as: No U-turn," is elegantly displayed on the LCD screen for the user to see.
7. **Continuation or Completion**: The system's design accommodates the evaluation of multiple images.

in a session. After each classification, the system reverts to its initial state of monitoring the push button. This cycle persists, allowing for successive image evaluations upon each button press. This interactive loop empowers the user to control the pace of image processing and is indicative of the system's design, which prioritizes user engagement and control. The process is designed to continue until all predetermined images have been classified, after which the system will display a completion message. This message not only signifies the end of the session but also assures the user of the system's readiness for the next sequence of operations.

In conclusion, the hardware interaction within the TSR system is characterized by its responsiveness to user inputs, the precision of its image processing protocols, and the clarity of its output. The images provided offer a visual narrative of the system in action, from the moment of activation to the conclusion of a classification task, encapsulating the user-friendly and interactive nature of the TSR system's design.

4.2 Result Analysis

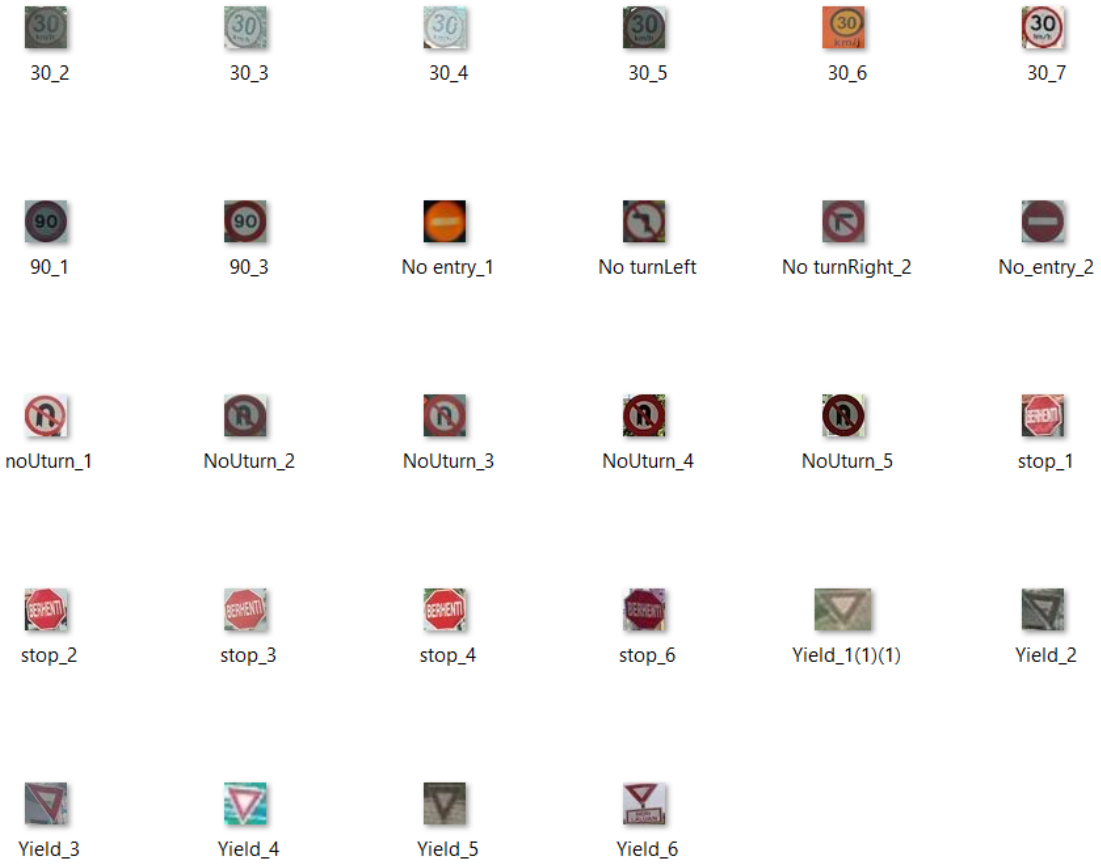


Figure 4-9 Test image dataset which a separate with training dataset.

Table 4-1 List of all the test image dataset output.

Image	Predicted	Unpredicted	Predict as
30_2	/		
30_3	/		
30_4	/		
30_5	/		
30_6	/		
30_7	/		
90_1		X	Speed limit 30mp/h
90_3		X	Speed limit 30mp/h
No entry_1		X	Stop

No turnLeft		X	No uturn
No turnRight_2		X	No uturn
No entry_2		X	Stop
noUturn_1	/		
NoUturn_2	/		
NoUturn_3	/		
NoUturn_4	/		
NoUturn_5	/		
Stop_1	/		
Stop_2	/		
Stop_3	/		
Stop_4	/		
Stop_6	/		
Yield_1(1)(1)	/		
Yield_2	/		
Yield_3	/		
Yield_4	/		
Yield_5	/		
Yield_6	/		

The test dataset, as illustrated in Figure 4-9, comprised a diverse range of traffic signs, including speed limit signs of various denominations, prohibitory signs such as 'No entry' and 'No turn', as well as imperative signs like 'Stop' and 'Yield'. These images were distinct from the ones used during the training phase to ensure an unbiased evaluation of the system's generalization capabilities.

Table 4-1 list of all the test image dataset output. The summarization of the classification outcomes is as follows:

- The system demonstrated a high accuracy rate for speed limit signs labeled '30', consistently recognizing and classifying them correctly across multiple test cases.

- However, certain instances of misclassification were noted for signs other than '30'. For example, some '90' speed limit signs were incorrectly identified as '30', and a 'No entry' sign was mistaken for a 'Stop' sign.
- The prohibitory signs such as 'No turn left' and 'No turn right' posed a challenge for the system, with misclassifications indicating that these signs were confused with No U-turn signs in the dataset.
- Signs such as 'No U-turn' and 'Stop' were recognized correctly, indicating a robust understanding of these categories by the model.
- The system did not predict any classification for signs that did not belong to the four categories it was trained on, due to the model's output tensor being explicitly limited to these classes.

In summary, the TSR system showcased commendable proficiency in recognizing signs within its defined scope but faced limitations when confronted with signs outside of its training parameters.

4.3 Limitation

4.3.1 Delivering or Inserting Test Images for Prediction

One of the limitations of the project is the convoluted mechanism required to deliver or insert test images for prediction in the Traffic Sign Recognition (TSR) system. Currently, the process relies on multiple online platforms, including Google Drive and Colab, which introduces several preliminary steps before an image can be processed. This method is not optimal for real-time applications and hinders the system's efficiency. A proposed enhancement is to establish direct communication between the Cyclone V E board and a convenient device, utilizing the board's serial communication peripherals to streamline the transfer and processing of image data. This would not only simplify the setup but also enhance the system's responsiveness and user-friendliness.

4.3.2 Prediction out of 4 Classes (Yield, Speed Limit 30mp/h, Stop, No Uturn)

Considering the strategic decision made during the development of the Traffic Sign Recognition (TSR) system, the result analysis must be viewed through the lens of the project's scope and the constraints placed upon the neural network model during its training phase.

Focused Classification Scope: The TSR system was meticulously designed with a specific intent to recognize and classify four distinct categories of traffic signs. This decision was underpinned by a concern that introducing a broader range of classes, including a generic 'other' category, could potentially dilute the model's ability to precisely identify the core classes of interest. The CNN was therefore tailored to generate four output tensors, each corresponding to one of the four selected traffic sign classes.

Impact on Detection Capability: This deliberate limitation has a direct impact on the system's detection capabilities. Signs that do not fall within the four defined classes are inherently unrecognized by the system, not due to a deficiency in the model's learning capacity, but because of the defined project scope. In essence, the model was not provided with the knowledge or the means to identify signs beyond its trained categories, and as a result, these signs remain undetected and unclassified.

Rationale Behind the Decision: The rationale for this approach was grounded in the principle of precision over generalization. By constraining the output tensor to four classes, the project aimed to enhance the model's ability to distinguish between the chosen categories with a higher degree of accuracy. It was a calculated trade-off to avoid the potential risk of the model misclassifying signs from the four primary classes as 'other', which could introduce uncertainty into the system's predictions.

Reflection on the Results: Reflecting on the results, it becomes evident that the system performs with high fidelity within the predetermined scope, accurately classifying the four types of traffic signs it was trained to recognize. The absence of an 'other' category means that the system does not attempt to classify signs outside of these categories, leading to instances where it does not provide a classification. This is

not a limitation but a feature of the system, ensuring that when it does make a classification, it does so with the utmost confidence.

Discussion on Model Strategy: This focused strategy has implications for the real-world application of the TSR system. It implies that while the system excels within its operational domain, it is not equipped to handle signs outside of this domain. This limitation is aligned with the project's objectives and scope, which centered on creating a model with a high degree of accuracy for a specific subset of traffic signs, rather than a broader but potentially less precise recognition system.

In summary, the TSR system was designed with a clear vision and a precise scope, leading to a result analysis that reflects the system's specialized capability. The decision to limit the output tensor to four classes was informed by a focus on the reliability of the CNN traffic sign recognition to be implement in the Cyclone V E FPGA board, which it is within the scope of the project. While this choice inherently limits the system's ability to recognize signs outside of its scope, it ensures that within the scope, the system's performance is robust and dependable.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Conclusion

This research successfully implemented a Traffic Sign Recognition (TSR) system on the Intel FPGA Cyclone V E Development Kit. The system is capable of accurately recognizing Malaysian road signs, highlighting the potential of FPGA-based embedded systems in intelligent transport applications. Despite the computational constraints of the FPGA's softcore processor, the project achieved a significant milestone by converting a Keras-trained CNN model into a deployable C source code for real-time sign recognition. The methodology's efficacy is evident in the system's performance, proving that with dedication and skill, complex algorithms can be executed efficiently on an FPGA platform.

5.2 Future Works

The future trajectory of this Traffic Sign Recognition (TSR) system is anchored in enhancing real-time operational capabilities and broadening the spectrum of recognizable traffic signs. To this end, the system can benefit substantially from the implementation of direct hardware communication protocols. Such an upgrade would streamline the image data transfer process, diminishing the current reliance on multi-step, external online platforms and advancing the system's efficiency.

Moreover, the expansion of the system's recognition capabilities to include additional sign classes is recommended. This could be realized through the introduction of a hierarchical classification system, enabling the model to discern between 'known' and 'unknown' categories before proceeding to detailed classification. This refinement would necessitate improvements in the system's robustness to diverse environmental factors, and optimizations focused on reducing power consumption and increasing processing speeds.

These proposed developments are aimed at extending the system's utility and applicability, ensuring that the TSR system evolves to meet the increasing demands of autonomous vehicular technologies and contributes meaningfully to the advancement of road safety.

REFERENCES

[1] H. Irmak, “REAL TIME TRAFFIC SIGN RECOGNITION SYSTEM ON FPGA A THESIS SUBMITTED TO THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF MIDDLE EAST TECHNICAL UNIVERSITY,” 2010.

[2] R. Hmida, A. Ben Abdelali, and A. Mtibaa, “Hardware implementation and validation of a traffic road sign detection and identification system,” in *Journal of Real-Time Image Processing*, Springer Verlag, Jun. 2018, pp. 13–30. doi: 10.1007/s11554-016-0579-x.

[3] A. Møgelmose, M. M. Trivedi, and T. B. Moeslund, “Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 4, pp. 1484–1497, 2012, doi: 10.1109/TITS.2012.2209421.

[4] A. Soetedjo and I. Komang Somawirata, “AN EFFICIENT ALGORITHM FOR IMPLEMENTING TRAFFIC SIGN DETECTION ON LOW COST EMBEDDED SYSTEM,” 2018.

[5] C. Souani, H. Faiedh, and K. Besbes, “Efficient algorithm for automatic road sign recognition and its hardware implementation,” *J Real Time Image Process*, vol. 9, no. 1, pp. 79–93, 2014, doi: 10.1007/s11554-013-0348-z.

[6] “Reference Manual Cyclone V E FPGA Development Board Feedback Subscribe Cyclone V E FPGA Development Board Reference Manual,” 2017. [Online]. Available: www.altera.com

[7] I. Corporation, “Nios® II Processor Reference Guide; Nios® II Processor Reference Guide.”

[8] W. Farhat, S. Sghaier, H. Faiedh, and C. Souani, “Design of efficient embedded system for road sign recognition,” *J Ambient Intell Humaniz Comput*, vol. 10, no. 2, pp. 491–507, Feb. 2019, doi: 10.1007/s12652-017-0673-3.

[9] Y. Han, K. Virupakshappa, E. V. S. Pinto, and E. Oruklu, “Hardware/software co-design of a traffic sign recognition system using zynq FPGas,” *Electronics* , vol. 4, no. 4, pp. 1062–1089, Dec. 2015, doi: 10.3390/electronics4041062.

[10] S. Waite and E. Oruklu, “FPGA-Based Traffic Sign Recognition for Advanced Driver Assistance Systems,” 2012, doi: 10.4236/jtts.2012.

[11] M. Á. García-Garrido, M. Á. Sotelo, and E. Martín-Gorostiza, “Fast traffic sign detection and recognition under changing lighting conditions,” in *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, Institute of Electrical and Electronics Engineers Inc., 2006, pp. 811–816. doi: 10.1109/itsc.2006.1706843.

[12] P. Viola and M. Jones, “Rapid Object Detection using a Boosted Cascade of Simple Features,” 2001.

[13] N. Dalal and B. Triggs, “Histograms of Oriented Gradients for Human Detection,” 2005. [Online]. Available: http://lear.inrialpes.fr

[14] W. Wang, H. Gao, X. Chen, Z. Yu, and M. Jiang, “Traffic sign detection based on Haar and adaBoost classifier,” in *Journal of Physics: Conference Series*, IOP Publishing Ltd, Apr. 2021. doi: 10.1088/1742-6596/1848/1/012091.

[15] Y. Freund and R. E. Schapire, “Experiments with a New Boosting Algorithm DRAFT-PLEASE DO NOT DISTRIBUTE,” 1996. [Online]. Available: http://www.research.att.com/orgs/ssr/people/fyoav,schapireg/

[16] Reinaldo, N. Manurung, J. I. Simbolon, and Christnatalis, “Traffic sign detection using histogram of oriented gradients and max margin object detection,” in *Journal of Physics: Conference Series*, Institute of Physics Publishing, Sep. 2019. doi: 10.1088/1742-6596/1230/1/012098.

[17] S. Tian *et al.*, “Multilingual scene character recognition with co-occurrence of histogram of oriented gradients,” *Pattern Recognit*, vol. 51, pp. 125–134, Mar. 2016, doi: 10.1016/j.patcog.2015.07.009.

[18] P. Carcagní, D. Cazzato, M. Del Coco, M. Leo, G. Pioggia, and C. Distante, “Real-time gender based behavior system for human-robot interaction,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 2014, pp. 74–83. doi: 10.1007/978-3-319-11973-1_8.

[19] A. R. Dubey, N. Shukla, and D. Kumar, “Detection and Classification of Road Signs Using HOG-SVM Method,” 2020, pp. 49–56. doi: 10.1007/978-981-13-9683-0_6.

[20] C. Cortes, V. Vapnik, and L. Saitta, “Support-Vector Networks Editor,” Kluwer Academic Publishers, 1995.

[21] L. Breiman, “Random Forests,” 2001.

[22] A. Ellahyani, M. El Ansari, and I. El Jaafari, “Traffic sign detection and recognition based on random forests,” *Applied Soft Computing Journal*, vol. 46, pp. 805–815, Sep. 2016, doi: 10.1016/j.asoc.2015.12.041.

[23] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553. Nature Publishing Group, pp. 436–444, May 27, 2015. doi: 10.1038/nature14539.

[24] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation.” [Online]. Available: http://arxiv.

[25] A. Kobbane, K. Ibrahimi, M. Y. Hadi, IEEE Communications Society, Institute of Electrical and Electronics Engineers. Morocco Section, and Institute of Electrical and Electronics Engineers, *Proceedings, 2017 the International Conference on Wireless Networks and Mobile Communications (WINCOM) : November 01-04, 2017, Rabat, Morocco*.

[26] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation.” [Online]. Available: http://arxiv.

[27] K. Ramasubramanian and A. Singh, “Deep Learning Using Keras and TensorFlow,” in *Machine Learning Using R*, Berkeley, CA: Apress, 2019, pp. 667–688. doi: 10.1007/978-1-4842-4215-5_11.

[28] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” Mar. 2016, [Online]. Available: http://arxiv.org/abs/1603.04467

[29] I. Corporation, “Nios® II Software Developer Handbook; Nios® II Software Developer Handbook.”

[30] A. Corporation, “Reference Manual Cyclone V E FPGA Development Board Feedback Subscribe Cyclone V E FPGA Development Board Reference Manual,” 2013. [Online]. Available: www.altera.com

[31] A. Madani and R. Yusof, “Malaysian Traffic Sign Dataset for Traffic Sign Detection and Recognition Systems”, [Online]. Available: http://vcari.net/cairo/downloads/MTSD.part5.rar

[32] M. Lebedev and P. Belecky, “A Survey of Open-source Tools for FPGA-based Inference of Artificial Neural Networks,” in *Proceedings - 2021 Ivannikov Memorial Workshop, IVMEM 2021*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 50–56. doi: 10.1109/IVMEM53963.2021.00015.

Appendix A Development of CNN Model for Traffic Sign Classification Algorithm

```
from google.colab import drive
drive.mount('/content/drive')

pip install numpy
pip install Pillow

import os
from keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten,
Dense
from keras.callbacks import ModelCheckpoint
from keras import backend as K
import cv2
from google.colab.patches import cv2_imshow # render the
image
import time
from scipy import ndimage
import scipy

!ls '/content/drive/MyDrive/TSR'
train_data_dir = '/content/drive/MyDrive/TSR/Train'
validation_data_dir =
'/content/drive/MyDrive/TSR/Validate'

# Specify the directory containing the images
directory = '/content/drive/MyDrive/TSR/Train/NoUturn'

# Get a list of all the image files in the directory
image_files = [f for f in os.listdir(directory) if
f.endswith('.jpg') or f.endswith('.png')]

# Set the number of images per row
images_per_row = 4

# Calculate the number of rows needed
num_rows = len(image_files) // images_per_row
if len(image_files) % images_per_row != 0:
    num_rows += 1

# Create a subplots grid with the specified number of
rows and columns
fig, axs = plt.subplots(num_rows, images_per_row,
figsize=(15, 15))
```

```
# Iterate over the image files and display them in a row
for i, filename in enumerate(image_files):
    filepath = os.path.join(directory, filename)

    # Read the image
    im = cv2.imread(filepath)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

    # Calculate the position in the subplots grid
    row = i // images_per_row
    col = i % images_per_row

    # Display the image in the corresponding subplot
    axs[row, col].imshow(im)
    axs[row, col].axis('off')  # hide axis

# Adjust the spacing between subplots
plt.subplots_adjust(wspace=0.05, hspace=0.05)

# Show the plot
plt.show()

# Specify the directory containing the images
directory =
'/content/drive/MyDrive/TSR/Train/SpeedLimit30'

# Get a list of all the image files in the directory
image_files = [f for f in os.listdir(directory) if
f.endswith('.jpg') or f.endswith('.png')]

# Set the number of images per row
images_per_row = 4

# Calculate the number of rows needed
num_rows = len(image_files) // images_per_row
if len(image_files) % images_per_row != 0:
    num_rows += 1

# Create a subplots grid with the specified number of
rows and columns
fig, axs = plt.subplots(num_rows, images_per_row,
figsize=(15, 15))

# Iterate over the image files and display them in a row
for i, filename in enumerate(image_files):
    filepath = os.path.join(directory, filename)

    # Read the image
    im = cv2.imread(filepath)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
```

```
    # Calculate the position in the subplots grid
    row = i // images_per_row
    col = i % images_per_row

    # Display the image in the corresponding subplot
    axs[row, col].imshow(im)
    axs[row, col].axis('off')  # hide axis

# Adjust the spacing between subplots
plt.subplots_adjust(wspace=0.05, hspace=0.05)

# Show the plot
plt.show()

# Specify the directory containing the images
directory = '/content/drive/MyDrive/TSR/Train/Stop'

# Get a list of all the image files in the directory
image_files = [f for f in os.listdir(directory) if
f.endswith('.jpg') or f.endswith('.png')]

# Set the number of images per row
images_per_row = 4

# Calculate the number of rows needed
num_rows = len(image_files) // images_per_row
if len(image_files) % images_per_row != 0:
    num_rows += 1

# Create a subplots grid with the specified number of
rows and columns
fig, axs = plt.subplots(num_rows, images_per_row,
figsize=(15, 15))

# Iterate over the image files and display them in a row
for i, filename in enumerate(image_files):
    filepath = os.path.join(directory, filename)

    # Read the image
    im = cv2.imread(filepath)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

    # Calculate the position in the subplots grid
    row = i // images_per_row
    col = i % images_per_row

    # Display the image in the corresponding subplot
    axs[row, col].imshow(im)
    axs[row, col].axis('off')  # hide axis
```

```
# Adjust the spacing between subplots
plt.subplots_adjust(wspace=0.05, hspace=0.05)

# Show the plot
plt.show()

# Specify the directory containing the images
directory = '/content/drive/MyDrive/TSR/Train/Yield'

# Get a list of all the image files in the directory
image_files = [f for f in os.listdir(directory) if
f.endswith('.jpg') or f.endswith('.png')]

# Set the number of images per row
images_per_row = 4

# Calculate the number of rows needed
num_rows = len(image_files) // images_per_row
if len(image_files) % images_per_row != 0:
    num_rows += 1

# Create a subplots grid with the specified number of
rows and columns
fig, axs = plt.subplots(num_rows, images_per_row,
figsize=(15, 15))

# Iterate over the image files and display them in a row
for i, filename in enumerate(image_files):
    filepath = os.path.join(directory, filename)

    # Read the image
    im = cv2.imread(filepath)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

    # Calculate the position in the subplots grid
    row = i // images_per_row
    col = i % images_per_row

    # Display the image in the corresponding subplot
    axs[row, col].imshow(im)
    axs[row, col].axis('off')  # hide axis

# Adjust the spacing between subplots
plt.subplots_adjust(wspace=0.05, hspace=0.05)

# Show the plot
plt.show()

#Model's Parameter
img_width, img_height = 100, 100 # dimensions of our
images.
```

```
nb_train_samples = 64
nb_validation_samples = 53
epochs = 50
batch_size = 10
dim = 3

# Choosing Image Format
if K.image_data_format() == 'channels_first':
    input_shape = (dim, img_width, img_height)
else:
    input_shape = (img_width, img_height, dim)

def larger_model():
    model = Sequential()
    model.add(Conv2D(30, (5,5), input_shape=input_shape,
activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Conv2D(15, (3,3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50,activation='relu'))
    model.add(Dense(4, activation='softmax'))
    return model

# build the model
model = larger_model()

# this is the augmentation configuration we will use for
training
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    color_mode="rgb",
    class_mode='categorical')

# this is the augmentation configuration we will use for
testing:
# only rescaling
test_datagen = ImageDataGenerator(rescale=1. / 255)

validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
```

```
    target_size=(img_width, img_height),
    batch_size=batch_size,
    color_mode="rgb",
    class_mode='categorical')

# How to save augmented image in a new folder

# Initialize save path
save_here = '/content/drive/MyDrive/Colab
Notebooks/augment'

# Initialize file
files = os.listdir(train_data_dir)
print (files)

for f in files:
    print(f)

    # Initialize file
    animal_path = os.listdir(train_data_dir+"/"+f)
    print (animal_path)

    # Loop all image in file
    for i in animal_path:

        # Read an image
        image_path = train_data_dir+"/"+f+"/"+i
        image = cv2.imread(image_path)
        image = np.expand_dims(image, 0)

        # fit the original image
        train_datagen.fit(image)

        # Initialize save path with class name
        save_path = save_here +"/"+ f

        # Create folder if not exist
        if not os.path.exists(save_path):
            os.makedirs(save_path)

        # Split original image name to get initial name
        f_name = i.split('.')
        f_name = f_name[0]+"_"+f_name[1]

        # Loop to create and save 5 augmented image for 1
original image
        for x, val in zip(train_datagen.flow(image,
#image we chose
            save_to_dir=save_path,     #this is where we
figure out where to save
```

```
            save_prefix=('{}_aug'.format(f_name)),
# it will save the images as 'aug_0912' some number for
every new augmented image
            save_format='jpg'),range(5)) :     # here we
define a range because we want 5 augmented images
otherwise it will keep looping forever
            pass

# Compile Model
model.compile(optimizer='rmsprop',loss='categorical_cross
entropy',metrics=['accuracy'])
# Train Model
history = model.fit_generator(
    train_generator,
    steps_per_epoch=nb_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=nb_validation_samples // batch_size)

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation
Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.savefig('./foo.png')
plt.show()

# Specify the directory containing the images
directory = '/content/drive/MyDrive/TSR/Test'

# Get a list of all the image files in the directory
image_files = [f for f in os.listdir(directory) if
f.endswith('.jpg') or f.endswith('.png')]

# Iterate over the image files and perform prediction
```

```
for filename in image_files:
    filepath = os.path.join(directory, filename)

    sign = cv2.imread(filepath)
    cv2_imshow(sign)

    # Read and preprocess the image
    im = cv2.imread(filepath)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
    im = cv2.resize(im, (img_width, img_height))
    im = np.reshape(im, [1, img_width, img_height, dim])

    # Make the prediction
    predicted_probabilities = model.predict(im)
    predicted_labels = np.argmax(predicted_probabilities,
axis=1)

    # Display the prediction result
    if predicted_labels == 0:
        print(f"{filename}: Predicted as No Uturn")
    elif predicted_labels == 1:
        print(f"{filename}: Predicted as 30km/h")
    elif predicted_labels == 2:
        print(f"{filename}: Predicted as Stop")
    elif predicted_labels == 3:
        print(f"{filename}: Predicted as Yield")
    else:
        print(f"{filename}: Unable to determine the
prediction")
```

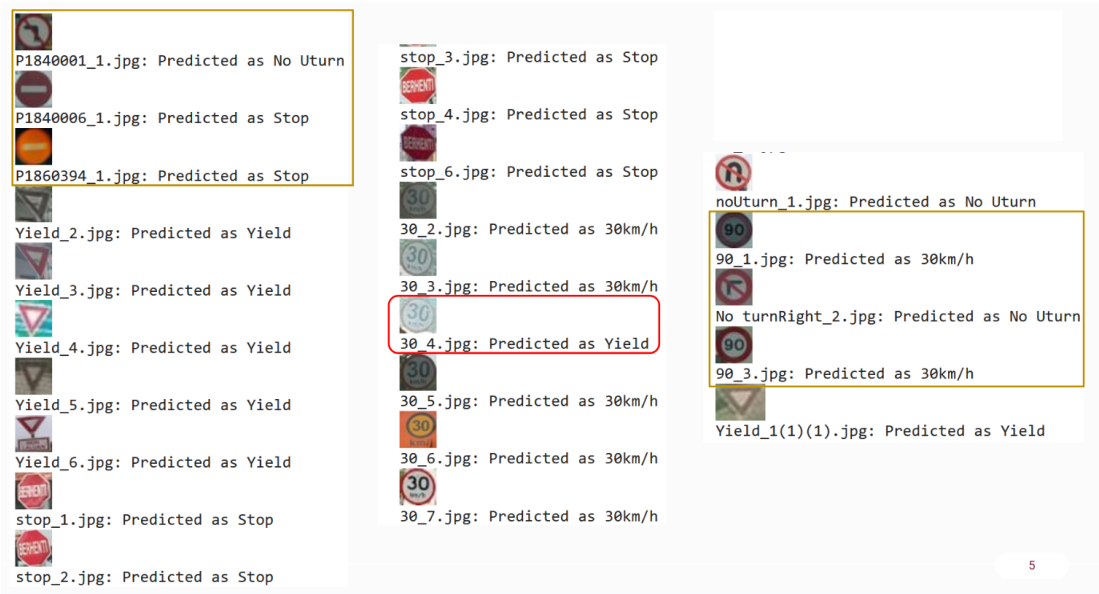


Figure A-5-1 Result of the test and verify of Keras trained CNN model.

Appendix B Model Conversion

Keras into ONNX Model

```
# Import tf2onnx
import tf2onnx

# Assuming 'model' is your Keras model that has been
defined and trained
# If not, make sure to define and train your Keras model
before running this cell

# Convert Keras model to ONNX format
onnx_model, _ = tf2onnx.convert.from_keras(model)

# Save the ONNX model to a file
onnx_filename = '/content/drive/MyDrive/Colab
Notebooks/TSR_phase1.onnx'
with open(onnx_filename, "wb") as f:
    f.write(onnx_model.SerializeToString())
```

Verify ONNX Model

```
import os
import cv2
import numpy as np
import onnxruntime
from tensorflow import keras
from google.colab.patches import cv2_imshow


# Load ONNX model
onnx_filename = '/content/drive/MyDrive/Colab
Notebooks/TSR_phase1.onnx'
sess = onnxruntime.InferenceSession(onnx_filename)

# Specify the directory containing the images
directory = '/content/drive/MyDrive/TSR/Test'

# Get a list of all the image files in the directory
image_files = [f for f in os.listdir(directory) if
f.endswith('.jpg') or f.endswith('.png')]

img_width, img_height = 32, 32  # Adjust the dimensions
based on your model's input size
dim = 3  # Assuming it's a color image (3 channels)
```

```
# Iterate over the image files and perform prediction
for filename in image_files:
    filepath = os.path.join(directory, filename)

    sign = cv2.imread(filepath)
    cv2_imshow(sign)

    # Read and preprocess the image
    im = cv2.imread(filepath)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
    im = cv2.resize(im, (img_width, img_height))
    im = np.reshape(im, [1, img_width, img_height, dim])

    # Make the prediction using ONNX Runtime
    input_name = sess.get_inputs()[0].name
    output_name = sess.get_outputs()[0].name
    predicted_probabilities = sess.run([output_name],
{input_name: im.astype(np.float32)})[0]
    predicted_labels = np.argmax(predicted_probabilities,
axis=1)

    # Display the prediction result
    if predicted_labels == 0:
        print(f"{filename}: Predicted as No Uturn")
    elif predicted_labels == 1:
        print(f"{filename}: Predicted as 30km/h")
    elif predicted_labels == 2:
        print(f"{filename}: Predicted as Stop")
    elif predicted_labels == 3:
        print(f"{filename}: Predicted as Yield")
    else:
        print(f"{filename}: Unable to determine the
prediction")
```

Figure B-5-2 Result of the test and verification of the ONNX model after conversion.

ONNX into C Source Code

```
[ ] !git clone https://github.com/kraiskil/onnx2c.git

[ ] !cd onnx2c && git submodule update --init

[ ] !mkdir -p onnx2c/build
    !cd onnx2c/build && cmake -DCMAKE_BUILD_TYPE=Release .. && make

[ ] !onnx2c/build/onnx2c "/content/drive/MyDrive/Colab Notebooks/TSR_phase1.onnx" > "/content/drive/MyDrive/model.c"

[ ] !cat /content/drive/My\ Drive/model.c
```

Figure B-5-3 Command to implement ONNX to C conversion.

Verify C Model

```
#include <iostream>
#include <vector>
#include <filesystem>
#include "CImg.h"
#include "model.c"  // Directly include the source file

namespace fs = std::filesystem;
using namespace cimg_library;

int main() {
    // Specify the directory containing the images
    const std::string directory = "Test";

    // Vector to hold image file names
```

```
    std::vector<std::string> image_files;

    // Iterate over the directory and get all .jpg and
.png files
    for (const auto& entry :
fs::directory_iterator(directory)) {
        if (entry.is_regular_file()) {
            auto path = entry.path();
            if (path.extension() == ".jpg" ||
path.extension() == ".png") {
                image_files.push_back(path.string());
            }
        }
    }

    // Process each image file
    for (const auto& image_path : image_files) {
        // Load the image
        CImg<unsigned char> image(image_path.c_str());
        if (image.width() != 32 || image.height() != 32
|| image.spectrum() != 3) {
            image.resize(32, 32, 1, 3);  // Resize to
32x32 and ensure 3 channels (RGB)
        }

        // Prepare the input tensor
        float input_tensor[1][32][32][3];
        cimg_forC(image, c) {
            cimg_forXY(image, x, y) {
                input_tensor[0][y][x][c] = image(x, y, 0,
c) / 255.f;  // Normalize pixel values
            }
        }

        // Output tensor
        float output_tensor[1][4];

        // Perform inference
        entry(input_tensor, output_tensor);

        // Determine the predicted class by finding the
index of the max value
        int predicted_class = 0;
        for (int i = 1; i < 4; ++i) {
            if (output_tensor[0][i] >
output_tensor[0][predicted_class]) {
                predicted_class = i;
            }
        }

        // Print the prediction
```

```
        std::cout << image_path << ": Predicted as ";
        switch (predicted_class) {
            case 0: std::cout << "No U-turn"; break;
            case 1: std::cout << "30km/h"; break;
            case 2: std::cout << "Stop"; break;
            case 3: std::cout << "Yield"; break;
            default: std::cout << "Unable to determine
the prediction"; break;
        }
        std::cout << std::endl;
    }

    return 0;
}
```

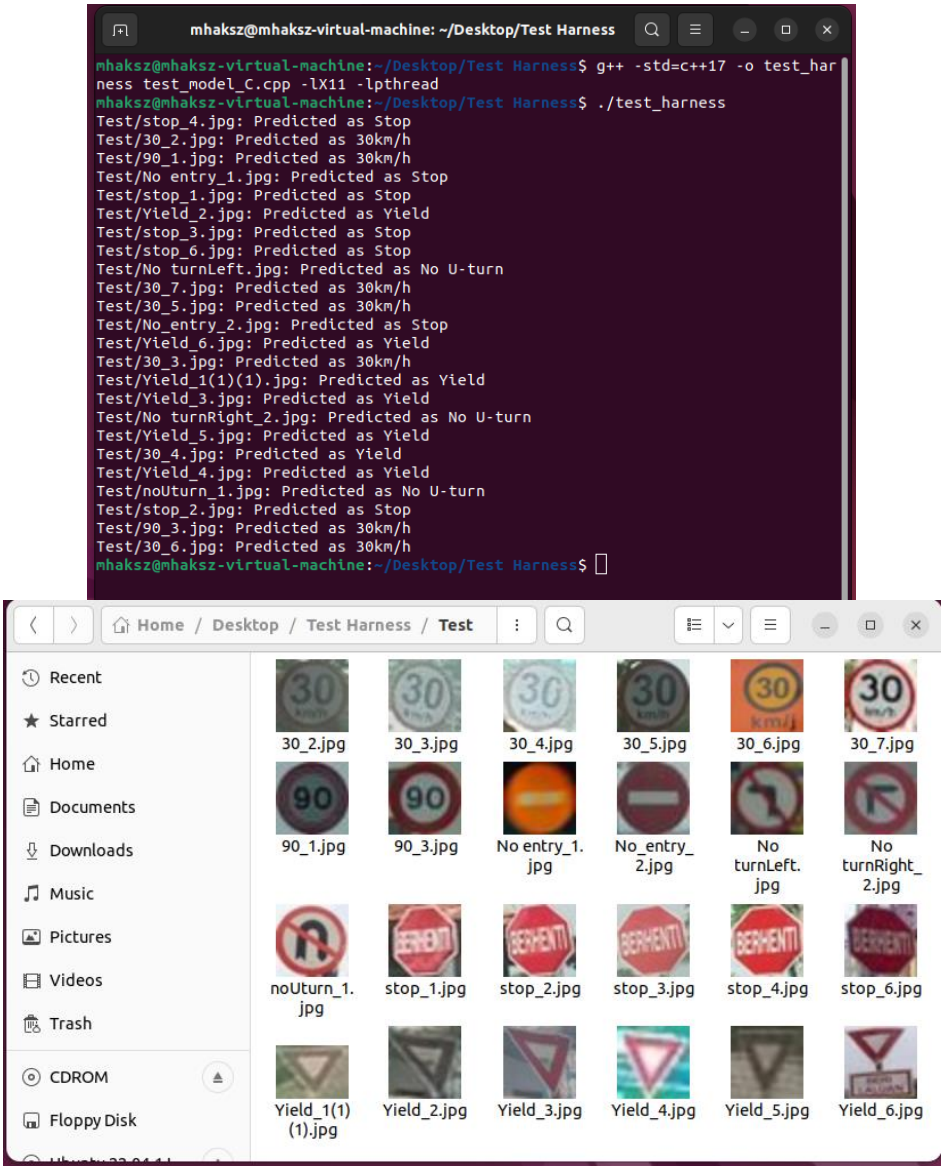


Figure B-5-4 Result of the test and verification of the C model after conversion.

Appendix C NIOS II SBT for Eclipse Project Application

Main Source Application Code (Test_harness.c)

```
#include <stdio.h>
#include "sys/alt_stdio.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include "io.h"
#include "model.h" // Ensure model.c contains the entry
function
#include "image_data.h" // Contains the raw_image_data
array

#define IMAGE_SIZE (32 * 32 * 3) // Size for a 32x32 RGB
image
extern const uint8_t
raw_image_data[NUMBER_OF_IMAGES][IMAGE_SIZE]; // Defined
in image_data.c
float output_tensor[1][4]; // Buffer for the output from
the neural network

// Function prototypes
//int classify_image(const float output_tensor[1][4]);
void display_results(int classification);
void simple_delay(unsigned int delay);
void SendCommand(alt_u8 cmd);    // Send command to lcd
void SendData(alt_u8 data);      // Send one character to
lcd
void SendMessage(char *msg);     // Send a string to lcd
void display_start();

int main() {
    alt_putstr("Starting the application...\n");
    display_start();

    for (int img_idx = 0; img_idx < NUMBER_OF_IMAGES;
++img_idx) {
     // Wait for push button to be pressed
     alt_putstr("Monitoring push button...\n");
     while (1) {
                 int button_data =
IORD_ALTERA_AVALON_PIO_DATA(PB_BASE) & 0x3; // Assuming
you are using the first button
                 if (button_data == 1) { // Check if the
first button is pressed
                     alt_putstr("button pressed...\n");
                      break; // Exit the waiting loop
                 }
             }
```

```
     // Debounce delay to avoid multiple reads from a
single press
     simple_delay(100000); // Adjust delay as needed for
your system
        float formatted_input[1][32][32][3]; // Array to
hold formatted input
        alt_putstr("Count loop 1...\n");
        for (int i = 0; i < IMAGE_SIZE; ++i) {
            formatted_input[0][i / 96][(i % 96) / 3][i %
3] = raw_image_data[img_idx][i] / 255.0f;
        }

        float output_tensor[1][4]; // Buffer for the
output from the neural network
        entry(formatted_input, output_tensor); // Use
formatted input for inference

        int predicted_class = 0;
        float max_value = output_tensor[0][0];
        for (int i = 1; i < 4; ++i) {
            if (output_tensor[0][i] > max_value) {
                max_value = output_tensor[0][i];
                predicted_class = i;

            }
        }
        display_results(predicted_class);
        alt_putstr("Classification completed.\n");
        char predicted_class_str[12];
        sprintf(predicted_class_str, "%d",
predicted_class);
        alt_putstr("Predicted Class: ");
        alt_putstr(predicted_class_str);
        alt_putstr("\n");


    }

    alt_putstr("Application finished.\n");
    return 0;
}


void display_start(){
     alt_putstr("Hello from Nios II!\n");

         // Initialize lcd
         SendCommand(0x0038);  // Function set: 8 bit, 2
lines 5*8 dots
         SendCommand(0x000F);  // Display on, cursor on,
cursor blinking
```

```
          SendCommand(0x0001);  // Display clear
          SendCommand(0x0006);  // Entry mode: right-
moving cursor (address increment), no display shift

          // Write first line message to lcd
          SendMessage("Traffic Sign");

          // Change DDRAM location to 40H to map to the
second line
          SendCommand(0x00C0);  // Set DDRAM address to
40H

          // Write second line message to lcd
          SendMessage("Recognition");
}

void display_results(int classification) {
    // Set up the display or send commands to another
output as needed
    // Here, it's set up to use an LCD display connected
to the system
    alt_putstr("Displaying result on LCD...\n"); // Debug
message

    // Initialize the LCD display
    SendCommand(0x0038); // Function set: 8-bit, 2 line,
5x8 dots
    SendCommand(0x000F); // Display on, cursor blinking
    SendCommand(0x0001); // Clear display
    SendCommand(0x0006); // Entry mode: cursor moves to
the right
    // Write first line message to lcd
    SendMessage("Predicted as:");

    // Change DDRAM location to 40H to map to the second
line
    SendCommand(0x00C0);  // Set DDRAM address to 40H

    // Array of messages corresponding to the
classification indices
    char* messages[] = {
        "No U-turn",
        "Speed Limit 30",
        "Stop",
        "Yield",
        "Unknown Sign"
    };

    // Ensure the classification index is within bounds
before accessing the array
```

```
    char* message = (classification >= 0 &&
classification < 4) ? messages[classification] :
messages[4];

    // Send the message to the LCD
    SendMessage(message);
}

void simple_delay(unsigned int delay)
{
    for (unsigned int i = 0; i < delay; i++) {
        // This is a simple busy-wait loop.
        __asm__ volatile ("nop");
    }
}

void SendCommand(alt_u8 cmd) // bitbang
{
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0400
| cmd);
    simple_delay(10000); // Adjust the delay value as
needed
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0000
| cmd); // Enable
    simple_delay(10000);
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0400
| cmd);
    simple_delay(10000);
}

void SendData(alt_u8 data) // bitbang
{
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0600
| data);
    simple_delay(10000);
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0200
| data); // Enable
    simple_delay(10000);
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_BITBANG_BASE, 0X0600
| data);
    simple_delay(10000);
}


void SendMessage(char *msg)
{
    for (; *msg != 0; msg++)
    {
        SendData(*msg);
    }
}
```

model.h

```
#ifndef MODEL_H_
#define MODEL_H_

// Function prototype declaration
void entry(const float tensor_conv2d_input[1][32][32][3],
float tensor_dense_2[1][4]);

#endif /* MODEL_H_ */
```

image_data.h

```
#ifndef IMAGE_DATA_H_
#define IMAGE_DATA_H_

#include <stdint.h>

#define IMAGE_SIZE (32 * 32 * 3) // Size for a 32x32 RGB
image
#define NUMBER_OF_IMAGES 13 // Example number of images

extern const uint8_t
raw_image_data[NUMBER_OF_IMAGES][IMAGE_SIZE];

#endif /* IMAGE_DATA_H_ */
```

image_data.c

```
// my_image_data.c
#include "image_data.h"

const uint8_t raw_image_data[NUMBER_OF_IMAGES][
IMAGE_SIZE] = {
    {……},
    {……},
    .
    .
    {……};
```

model.c

// The code is too long to put. It consists of CNN trained model for our 4 classes (Yield, Stop, No Uturn, Speed Limit 30mp/h)
https://drive.google.com/file/d/1HjHpCjr83NW7mhVrDLGZcUK2UKtCSKHK/view?usp=drive_link