Implementation of Symbolic Controllers on FPGAs

Bachelor Thesis

Scientific work to obtain the degree
B.Sc. Electrical Engineering and Information Technology

Department of Electrical and Computer Engineering
Technical University of Munich.

Supervised by Prof. Majid Zamani
M.Sc. Mahmoud Khaled
Assistant Professorship of Hybrid Control Systems

Submitted by Maha Khalifa
Arcisstr. 21
80333 Munich

Filed on Munich , on 13.8.2018

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowledgement has been made in the text to all other material used

Maha Khalifa
12.8.2018

Acknowledgement

First and foremost, I would like to thank my supervisor Prof. Dr. Majid Zamani his support and guidance through my bachelor project. I would also like to thank my supervisor M.Sc.Mahmoud Khaled for his patience, support and help throughout the five months, I am very grateful for the professional and excellent learning experience he has provided me with.

I would also like to express my deepest and most sincere gratitude to my father Prof. Dr. Ahmed Rashad for everything he has done for me and all the love he always gives me. A well as my mother for her continuous care and support. In addition, my brothers Sherif and Hisham for always pushing me forward and having my back.

Last but not least, I would like to thank all of my friends who supported me in this phase, special thanks to my dearest friends Yomna Atef and Yomna Sherif for all the help, and massive support they gave me throughout the journey.

Abstract

In this thesis, we are concerned about BDDs which stand for binary decision diagrams, how to understand and construct them. We propose a new method of representing symbolic controllers represented as BDDs on FPGAs or microcontrollers in general.We deal with tools that convert these controllers into BDD and convert them to controllers .bdd text files. Our main aim throughout the thesis is to discuss step by step how we can understand these files, followed by implementation of the BDD based symbolic controllers on FPGA.

Contents

1 Introduction

Control systems are usually modeled by differential equations describing how physical phenomena can be influenced by certain control parameters or inputs. Although these models are very powerful when dealing with physical phenomena, they are less suitable to describe software and hardware interfacing the physical world. This has spurred a recent interest in describing control systems through symbolic models that are abstract descriptions of the continuous dynamics, where each symbol corresponds to an aggregate of continuous states in the continuous model. Since these symbolic models are of the same nature of the models used in computer science to describe software and hardware, they provided a unified language to study problems of control in which software and hardware interacts with the physical world. In this paper we show that every incrementally globally asymptotically stable nonlinear control system is approximately equivalent to symbolic model with a precision that can be chosen apriori. We also show that for digital controlled systems, in which inputs are piecewiseconstant, and under the stronger assumption of incremental inputtostate stability, the symbolic models can be obtained, based on a suitable quantization of the inputs. [9]

1.1 Thesis objective and scope

The main objective of the thesis is to represent symbolic controllers in form of Binary Decision diagrams to be injected in BDD2IMPLEMENT tools. So the BDD files are generated from scots to be injected in bdd2implement generate software implementations of BDD-based symbolic controllers.So throughout the whole thesis, we try to understand the bdd files that are generated with the help of CUDD library, how it encodes data, how to read it, how is binary decision diagrams are represented through this file. Afterwards , we construct a whole binary tree out of this binary decision diagrams represented as bdd files through.This tree is constructed through C++ code that can access these bdd files, read them and turn them into binary tree data structure to be saved as a memory.The main function of our C++ code is to read these files, saves them as a tree, traverses the tree.

2 Literature Review

2.1 Symbolic Controllers

Symbolic models are the right descriptions of continuous systems in which symbols represent aggregates of contiguous states. In the last few years, there has been a growing interest in the use of symbolic models as a tool for reducing the complexity of control designs. In fact, symbolic models enable the use of known algorithms in the context of supervisory control and algorithmic theory, for controller synthesis.

Since the nineteens, many researchers faced the problem of identifying classes of dynamical and control systems that admit symbolic models. Our main aim was to show that incrementally globally asymptotically stable nonlinear control systems with disturbances admit symbolic models. When specializing in these results in linear systems.

In recent years we have experienced the development of various symbolic techniques help in reducing the complexity of controller synthesis, these techniques are based on the idea that many states can be treated equally when synthesizing controllers and so can be replaced by a symbol. The models resulting from replacing equivalent states by symbols termed symbolic models which are typically simpler than the original ones, in a way that they have a lower number of states. In most cases, symbolic models could be constructed with a finite number of states.

Control systems are usually modeled by differential equations describing how physical phenomena can be influenced by certain control parameters or inputs. Although these models are very powerful when dealing with physical phenomena, they are less suitable to describe software and hardware interfacing with the world physically. This has promoted a recent interest in describing control systems through symbolic models that are fundamental descriptions of the continuous dynamics where each symbol corresponds to an aggregate of contiguous states in the continuous model.

Since these symbolic models are of the same properties of the models used in computer science, they provided a unified language to study problems of control in which both software and hardware interact with the physical world.
[9]

2.2 SCOTS Tool

SCOTS is an open source software tool for the synthesis of symbolic controllers for nonlinear control systems. It is implemented in C ++ and it comes with MATLAB interface to access the synthesized controller from inside MATLAB. The tool is supposed to be used and extended by researches in the area of formal methods for cyber- physical systems. SCOTS provides a basic implementation to symbolic synthesis. SCOTS provides an implementation of the various algorithms using two different data structures. One implementation is based on binary decision diagrams (BDD), which is our main area of interest in this Thesis.[12]

2.3 BDD2implement Tool

BDD2Implement is a C++ tool to generate hardware/software implementations of BDD-based symbolic controllers. Having the tools SCOTS and SENSE that generate BDD-based symbolic controllers of (networked) general nonlinear dynamical systems, BDD2Implement completes missing ring in the automatic synthesis technique.

BDD2Implement accepts static or dynamic determinized symbolic controllers in the form of BDD-files. The BDD files encode the controller dynamics as boolean functions. If the provided controller is not determinized, BDD2Implement provides a determinization of the controller.

Due to the technique used in BDD2Implement, the generated implementations are formal. This guarantees the generated codes are exactly achieving the behavior in the provided controllers. As a result, the whole development cycle SCOTS/SENSE/BDD2Implement is now formal.

BDD2Implement can generate codes in the following formats:

HARDWARE:

Verilog/VHDL modules

SOFTWARE:

C/C++ boolean-valued functions

BDD2Implement expects existing BDD-based symbolic controllers from SCOTS or SENSE.

It starts by converting the multi-output boolean functions inside BDDs to multi single-output functions. If the provided controller is not determinized, BDD2Implement provides a determinization of the controller using several possible determinization methods. For VHDL/Verilog, the boolean functions are dumped to the VHDL module contains the boolean functions as maps from input-port to output-port. For dynamic controllers, the HW module contains additional memory for the state of the controller. For C/C++, the boolean functions are dumped as C++ codes. The C/C++ compiler at implementer-side takes care of converting such boolean functions to machine codes.

It also expects information about the delay bounds in the NCS. Then, it operates within the symbolic abstraction of the plants to construct a symbolic abstraction for NCS. Plats symbolic models can be easily constructed using a tool like SCOTS.

BDD2Implement depends on the CUDD-3.0.0 library for manipulating BDDs, written by Fabio Somenzi here. The dddmp library is also used for reading and writing BDDs which already comes with CUDD.[1]

2.4 CUDD Library

CUDD stands for Colorado University Decision Diagram. It is a package for the manipulation of Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs) and Zero-suppressed Binary Decision Diagrams (ZDDs).

2.4.1 DDDMP Package

The DDDMP package inside CUDD library defines formats and rules to store DD on file. More, in particular, it contains a set of functions to dump (store and load) DDs and DD forests on f le in different formats. In the present implementation, BDDs (ROBDDs) and ADD (Algebraic Decision Diagram) of the CUDD package (version 2.3.0 or higher) are supported. These structures can be represented on files either in a text, binary, or CNF (DIMACS) formats. The main rules used are following rules: A file contains a single BDD/ADD or a forest of BDDs/ADD, i.e., a vector of Boolean functions. Integer indexes are used instead of pointers to reference nodes. BDD/ADD nodes are numbered with contiguous numbers, from 1 to NNodes (total number of nodes on a file). 0 is not used to allow negative indexes for complemented edges. A file contains a header, including several pieces of information about variables and roots of BDD functions, then the list of nodes. The header is always represented in text format (also for binary). BDDs, ADDs, and CNF share a similar format header. BDD/ADD nodes are listed following their numbering, which is produced by a post-order traversal, in such a way that a node is always listed after its Then/Else children.

Format

BDD dump files are composed of two sections: The header and the list of nodes. The header has a common (text) format, while the list of nodes is either in text or binary format, in our case it is text format. In text format nodes are represented with redundant information, where the main goal is readability, while the purpose of binary format is minimizing the overall storage size for BDD nodes. The header format is kept common to text and binary formats for sake of simplicity: No particular optimization is presently done on binary file headers, whose size is by far dominated by node lists in the case of large BDDs.In text mode nodes are listed on a text line basis. Each a node is represented as (First column→Node-index) (Second column →Var-extra-info) (Third column v→Var-internal-index) (Fourth column→Then-index) (Fifth column→Else-index) where all indexes are integer numbers. This format is redundant (due to the node ordering, Node-index is an incremental integer) but we keep it for readability. Var-extra-info (optional redundant eld) is either an integer (ID, PermID, or auxID) or a string (variable name). Var-internal-index is an internal variable index: Variables in the true support of the stored BDDs are numbered with ascending integers starting from 0, and follo (several thousands of DD nodes).

- Example:

```
.ver DDDMP-2.0   /*Dddmp version information*/
```

```
.mode A/* File mode (A for ASCII text, B for binary mode).*/
.varinfo 0 /*Var-extra-info (0: variable ID, 1: permID, 2: aux ID, 3: var
.nnodes 11   /*Total number of nodes in the file*/
.nvars 6   /*Number of variables of the writing DD manager.*/
.nsuppvars 6        /*Number of variables in the true support of the store
.ids 0 1 2 3 4 5       /*Variable IDs.*/
.permids 0 1 2 3 4 5          /*Variable permuted IDs.*/
.nroots 1              /*Number of BDD roots.*/
.rootids 11
.nodes
1 T 0 0 0
2 T 1 0 0
3 5 5 1 2
4 5 5 2 1
5 4 4 3 4
6 3 3 1 5
7 3 3 5 1
8 2 2 6 7
9 1 1 1 8
10 1 1 8 1
11 0 0 9 10
.end
```

3 Binary decision Diagrams

3.1 Binary Decision Diagrams

Binary decision diagram (BDD) or branching program is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression. Other data structures used to represent a Boolean function include negation normal form (NNF), and propositional directed acyclic graph (PDAG).

3.2 Binary trees

A binary tree is a tree data structure where each node has up to two child nodes, creating the branches of the tree. The two children are usually called the left and right nodes. Parent nodes are nodes with children, while child nodes may include references to their parents. There are multiple types of binary tree.

1. **Full Binary Tree**
   - If each node of the binary tree has either two children or no child at all, is said to be a Full Binary Tree.
   - Full binary tree is also called as Strictly Binary Tree.
   - Every node in the tree has either 0 or 2 children.
   - Full binary tree is used to represent mathematical expressions.
2. **Complete Binary Tree**
   - If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a Complete Binary Tree.
   - Complete binary tree is also called as Perfect Binary Tree.
   - In a complete binary tree, every internal node has exactly two children and all leaf nodes are at same level.
   - For example, at Level 2, there must be $22 = 4$ nodes and at Level 3 there must be $23 = 8$ nodes.
3. **Skewed Binary Tree**
   - If a tree which is dominated by left child node or right child node, is said to be a Skewed Binary Tree.
   - In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.
   - In a left-skewed tree, most of the nodes have the left child without corresponding right child.

- In a right-skewed tree, most of the nodes have the right child without corresponding left child.

4. **Extended binary tree**

   - Extended binary tree consists of replacing every null subtree of the original tree with special nodes.
   - Empty circle represents an internal node and filled circle represents the external node.
   - The nodes from the original tree are internal nodes and the special nodes are external nodes.
   - Every internal node in the extended binary tree has exactly two children and every external node is a leaf. It displays the result which is a complete binary tree.

5. **Advantages of Binary Trees**

   - Trees reflect structural relationships in the data
   - Trees are used to represent hierarchies
   - Trees provide an efficient insertion and searching
   - Trees are very flexible data, allowing to move subtrees around with minimum effort

3.3 Traversal

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds.

- depth first traversal
- breadth first traversal

There are three different types of depth-first traversals :

- **PreOrder traversal** - visit the parent first and then left and right children.We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be **A B D E C F G**

  Algorithm

1. **Step 1** Visit root node.
2. **Step 2** Recursively traverse left subtree.
3. **Step 3** Recursively traverse right subtree.

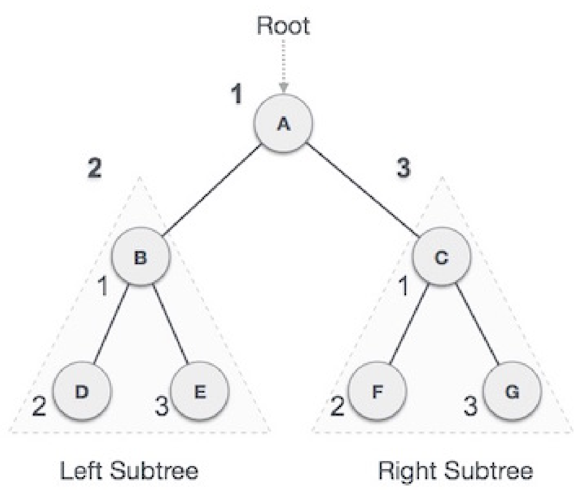

Figure 1: Pre-Order Traversal

- **InOrder traversal** - visit the left child, then the parent and the right child. We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be **D B E A F C G**

  Algorithm

  1. **Step 1** Recursively traverse left subtree.
  2. **Step 2** Visit root node.
  3. **Step 3** Recursively traverse right subtree.

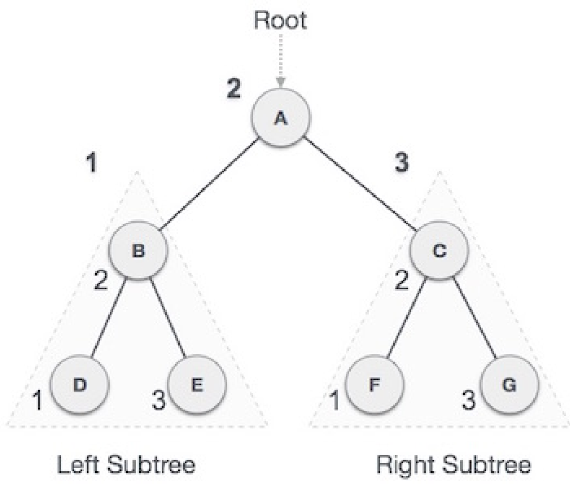

Figure 2: In-Order Traversal

- **PostOrder traversal** - visit the left child, then the right child and then the parent. We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be **D E B F G C A**

  Algorithm

  1. **Step 1** Recursively traverse left subtree.
  2. **Step 2** Recursively traverse right subtree.
  3. **Step 3** Visit root node.

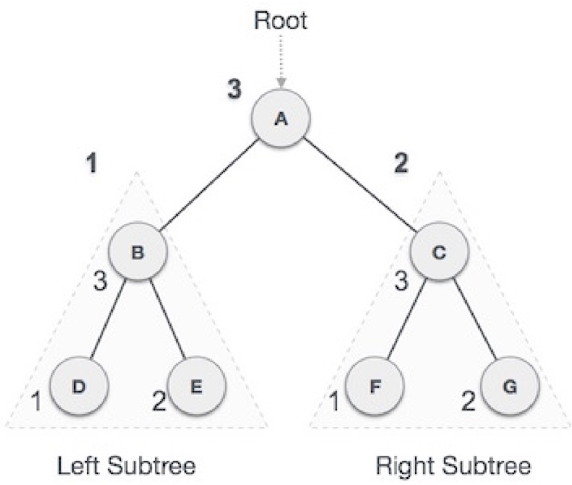

Figure 3: Post-Order Traversal

3.4 Boolean Functions

A Boolean function is described by an algebraic expression consisting of binary variables, the constants 0 and 1, and the logic operation symbols +,*. For a given set of values of the binary variables involved, the boolean function can have a value of 0 or 1. For example, the boolean function F= x' y + z is defined in terms of three binary variables x,y,z. The function is equal to 1 if x=0 and y=1 simultaneously or z=1. Every boolean function can be expressed by an algebraic expression, such as one mentioned above, or in terms of a Truth Table. A function may be expressed through several algebraic expressions, on account of them being logically equivalent, but there is only one unique truth table for every function. A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure. Circuit diagram for F.

A set of rules or Laws of Boolean Algebra expressions have been invented to help reduce the number of logic gates needed to perform a particular logic operation resulting in a list of functions or theorems known commonly as the Laws of Boolean Algebra.

Boolean Algebra is the mathematics we use to analyze digital gates and circuits. We can use these Laws of Boolean to both reduce and simplify a complex Boolean expression in an attempt to reduce the number of logic gates required. Boolean Algebra is, therefore, a system of mathematics based on a logic that has its own set of rules or laws which are used to define and reduce Boolean expressions.

The variables used in Boolean Algebra only have one of two possible values, a logic 0 and a logic 1 but an expression can have an infinite number of variables all labelled individually to represent inputs to the expression, For example, variables A, B, C etc, giving us a logical expression of A + B = C, but each variable can only be a 0 or a 1.

Examples of these individual laws of Boolean, rules, and theorems for Boolean Algebra are given in the following table.

Canonical and Standard Forms Any binary variable can take one of two forms, x or x'. A boolean function can be expressed in terms of n binary variables. If all the binary variables are combined together using the AND operation, then there are a total-combinations since each variable can take two forms. Each of the combinations is called a minterm or standard product. A minterm is represented by m_i where i is the decimal equivalent of the binary number the minterm is designated.

In a minterm, the binary variable is un-primed if the variable is 1 and it is primed if the variable is 0 i.e. if the minterm is xy' then that means x=1 and y=0. For example, for a boolean function in two variables the minterms are m_0=**x'y'** , m_1=**x' y** , m_2=**x' y** , m_3=**x y**

In a similar way, if the variables are combined together with OR operation, then the term obtained is called a maxterm or standard sum. A maxterm is represented by M_i where i is the decimal equivalent of the binary number the maxterm is designated. In a maxterm, the binary variable is un-primed if the variable is 0 and it is primed if the variable is 1 i.e. if the minterm is x'+y then that means x=1 and y=0. For example, for a boolean function in two variables the minterms are m_0=x+y,

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	x′y′z′	m_0	x + y + z	M_0
0	0	1	x′y′z	m_1	x + y + z′	M_1
0	1	0	x′yz′	m_2	x + y′+ z	M_2
0	1	1	x′yz	m_3	x + y′ + z′	M_3
1	0	0	xy′z′	m_4	x′ + y + z	M_4
1	0	1	xy′z	m_5	x′ + y + z′	M_5
1	1	0	xyz′	m_6	x′ + y′ + z	M_6
1	1	1	xyz	m_7	x′ + y′ + z′	M_7

Figure 4: Minterms and Maxterms for function in 3 variables

3.4.1 Boolean Identities

Double Complement Law
∼(∼ A)=A

Complement Law
A+A=1 (OR Form)
A.A=0 (AND Form)

Idempotent Law
A+A=A (OR Form)
A.A=A (AND Form)

Identity Law
A+0=A (OR Form)
A.1=A (AND Form)

Dominance Law
A+1=1 (OR Form)
A.0=0 (AND Form)

Commutative Law
A+B=B+A (OR Form)
A.B=B.A (AND Form)

Associative Law
A+(B+C)=(A+B)+C (OR Form)
A.(B.C)=(A.B).C (AND Form)

Absorption Law

A.(A+B)=A
A+(A.B)=A

Simplification Law

A.(A+B)=A.B
A+(A.B)=A+B

Distributive Law

A+(B.C)=(A+B).(A+C)
A.(B+C)=(A.B)+(A.C)

De-Morgan's Law

(A.B)=A+B
(A+B)=A.B

3.4.2 .bdd files

In order to understand more how .bdd file works, we needed CUDD library as well as to visualize these files for better understanding.

4 Methodology

In this chapter, we would like to discuss the Milestones of our project starts by CUDD library, binary decision diagrams, how they are represented, how boolean operations are applied to them. This is followed by controller.bdd files generated from scots, and finally, how a whole binary tree can be constructed out of any controller.bdd file, as well as representing it in memory in FPGAs or any other microcontrollers.

4.1 Milestone I: Boolean operations on BDDs using CUDD library

In this milestone, we focus more on CUDD library and how to use them in order to generate a name.bdd files, to understand the CUDD, we have performed simple boolean operations using binary decision diagrams.[5]

4.1.1 CUDD LIBRARY

The CUDD package is a package written in C for the manipulation of decision diagrams, it supports Binary Decision Diagrams (BDDs), Algebraic decision diagrams(ADDs), and Zero-Suppressed BDDs (ZDDs). The basic use of CUDD is as follows:

- Initialize a DdManager using Cudd—Init
- Create the DD
- Shut down the DdManager using Cudd–Quit(DdManager* ddmanager)

Sample code for the main program

The program below creates a single BDD variable

Creates a single BDD variable

```
#include "util.h"
#include "cudd.h"
int main (int argc, char *argv[])
{
    DdManager *gbm; /* Global BDD manager.*/
      char filename[30];
      gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0); /* Initialize a new BDD manager.*/
      DdNode *bdd = Cudd_bddNewVar(gbm); /*Create a new BDD variable*/
      Cudd_Ref(bdd); /*Increases the reference count of a node*/
      Cudd_Quit(gbm);
      return 0;
}
```

4.1.2 Boolean Operations using DDDMP package

The DDDMP package is a package inside CUDD library that defines formats and rules to store decision diagrams on file, so it is basically used for generating the **.bdd files**. After creating one node, we want to create a boolean function, which is full of multiple nodes and then observe the graph. The code here generates two files; the ".bdd file" and "the .dot file". The bdd file represents the binary decision diagram of the controller in text file.

We use the method **int Dddmp—cuddBddArrayStore** from DDDMP library that Dumps the argument array of BDDs to file. Dumping is either in text or binary form, but in our case, we choose the text format. The BDDs are stored in the fp (already open) file if not NULL. Otherwise, the file whose name is fname is opened in the write mode, the header has the same format for both textual and binary dump. Names are allowed for input variables (vnames) and represented functions (rnames). For the sake of generality, and because of dynamic variable, ordering both variables IDs and permuted IDs are included, new IDs are also supported (auxids).

Variables are identified with incremental numbers, according to their position in the support set. In text mode, an extra info may be added, chosen among the following options: name, ID, PermID, or an auxiliary (auxids). Since conversion from DD pointers to integers is required, DD nodes are temporarily removed from the unique hash table, this allows the use of the next field to store node IDs.

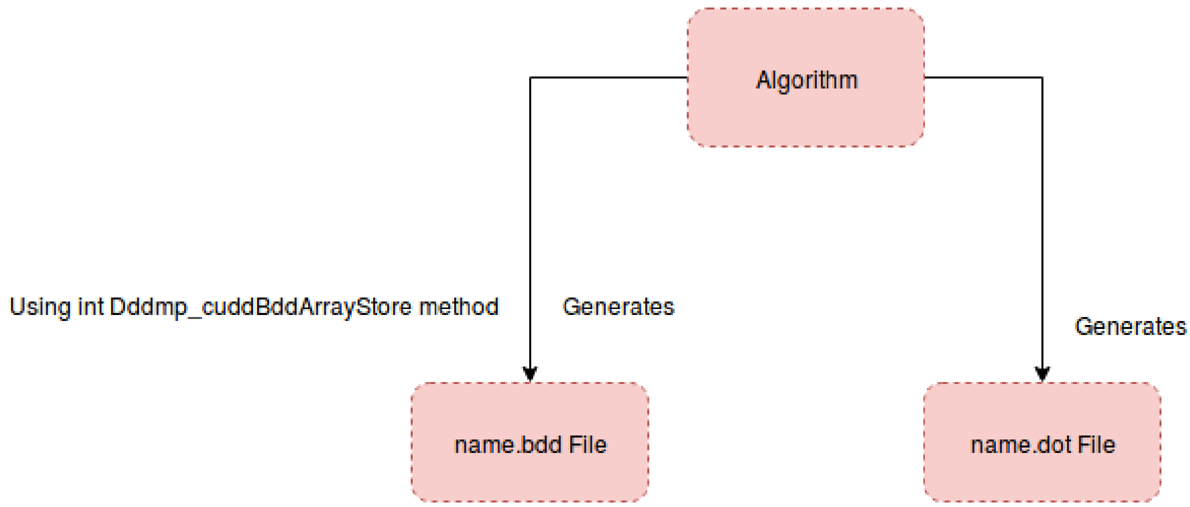

Figure 5: Milestone 1

```
int Dddmp_cuddBddArrayStore(
DdManager * dd, manager
char * ddname, dd name (or NULL)
int nroots, number of output BDD roots to be stored
DdNode ** f, array of BDD roots to be stored
char ** rootnames, array of root names (or NULL)
char ** varnames, array of variable names (or NULL)
```

```
    int * auxids, array of converted var IDs
    int  mode, storing mode selector
    Dddmp——VarInfoType  varinfo, extra info for variables in text mode
    char * fname, file name
    FILE * fp pointer to the store file
)
```

- Algorithm Explanation

  As mentioned in figure 5, our algorithm generates two output files, **name.bdd file** and **name.dot file**.To make sure everything is going in the correct path of our understanding to .bdd files, we would compare **name.bdd file** and **name.dot file** with our self drawn binary descision diagram of each function to make sure the two files are compatible.

  1. **Name.bdd file**

     This is a text file generated using CUDD Library, specifically DDDMP package, this file describes the binary decision diagram of any function in a form of columns of numbers, where each column represents a certain meaning. The first two rows of any name.bdd file are the terminal nodes which are either zeros or ones.

     – First Column:Node ID
       The name of the node.
     – Second Column:Node Level
       Represents the level of each node.**T** in the second column means Terminal. Thus, always nodeID 1 and 2 are the terminal nodes that represent in fact zero and one. To know which is zero and which is one, we would know this from the the third column that represents the Node Extra Info.
     – Third Column:Node Extra Info
       We only care about this column in the first two nodes only which are the first two rows only. We are interested in the values drawn located in the intersection of the first row (representing node 1)with the third column e.g.,(1,3), and the second row (representing node 2) with the third column e.g.,(2,3).
     – Fourth column:If-Node
       To reach the If-Node , a solid line is used.
     – Fifth column:Else-Node
       To reach the Else-Node , a dashed line is used.

  2. **Name.dot file**
     This is a visualisation to the Binary decision diagram created .

1. And Gate
   We would explain this example, and all other examples would follow the same instructions used. Referring to figure 7 and 32, we observe from And.bdd figure 7, there are five columns as discussed before, the first column contains nodes from 1 to 4 (means 4 nodes). Starting always from the last row of the file, thus we start with node 4.

   - Row 4 : The **Node Id** is 4 .The **Node Level** which is column 2 is 0. Therefore, in figure 32 we observe that Node 4 is in Level 0. The **If-Node** which is the fourth column is here 3. Thus, in figue 32, we observe Node 4 going to Node 3 with a solid line. The **Else-Node** is 2 , so a dashed line goes from Node 4 to Node 2 .

   - Row 3 : The **Node ID** is 3 .The **Node Level** which is column 2 is 1 , so in figure 32 ,we observe that Node 3 is in Level 1. The **If-Node** which is the fourth column is here 1 , so in figue 32 , we can observe Node 4 going to Node 1 with a solid line.The **Else-Node** is 2 , so a dashed line goes from Node 3 to Node 2 .

   - Row 2 : The **Node ID** is 2 .The **Node Level** which is column 2 is T which I concluded means Terminal nodes .The **Node Extra Info** which is column three here tells us whether this terminal node is 0 or 1 , here in this row , it is terminal zero ,means that **Node ID** 2 is implicitly/technically meaning that it is **Node ID** 0.The **If-Node** and **Else-Node** which is the fourth column and fifth column are here 0 which means they point to NULL as we do not have a node -Id with the value 0.

   - Row 1: The **Node ID** is 1 .The **Node Level** which is column 2 is T which I concluded means Terminal nodes.The **Node Extra Info** which is column three here tells us whether this terminal node is 0 or 1, here in this row, it is terminal zero, means that **Node ID** 1 is implicitly/technically meaning that it is **Node ID** 1.The **If-Node** and **Else-Node** which is the fourth column and the fifth column are here 0 which means they point to NULL as we do not have a node -Id with the value 0.

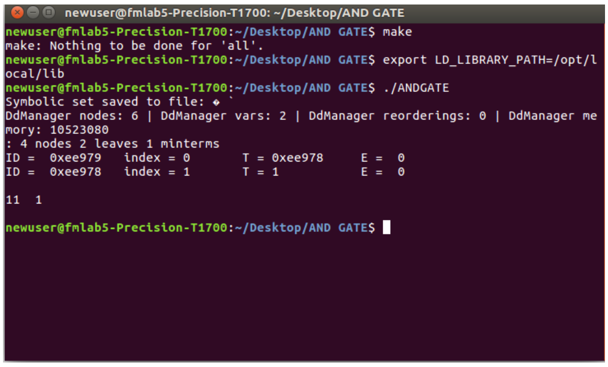

Figure 6: And Terminal Commands

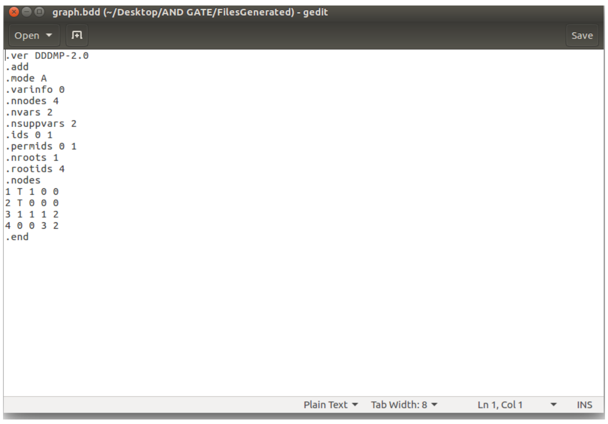

Figure 7: And.bdd File

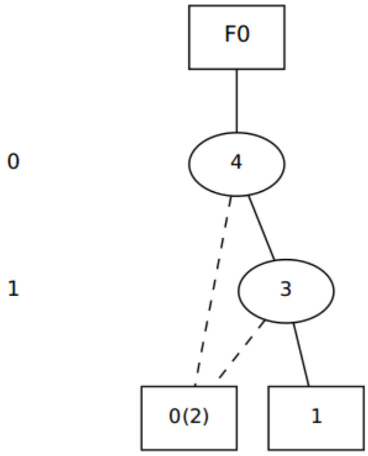

Figure 8: And.dot File

2. Nand Gate

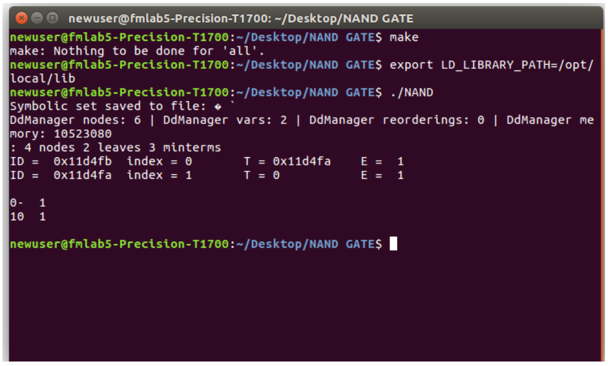

Figure 9: Nand Terminal Commands

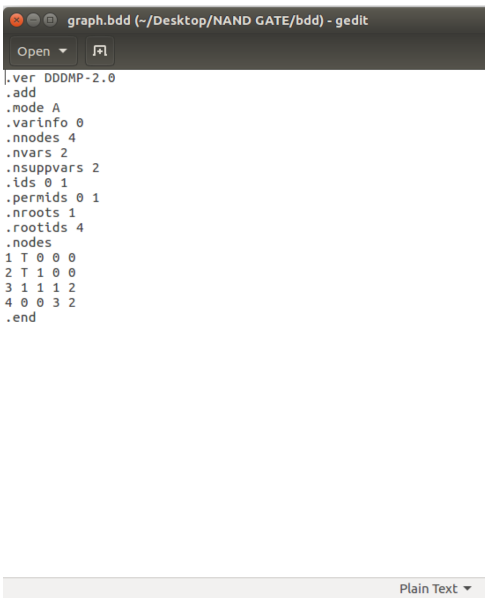

Figure 10: Nand.bdd

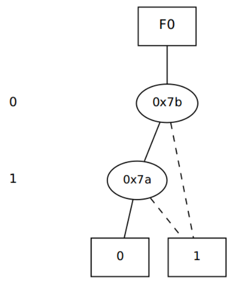

Figure 11: Nand.dot

3. Complex Function 1

Figure 12: Complex Function 1 Terminal Commands

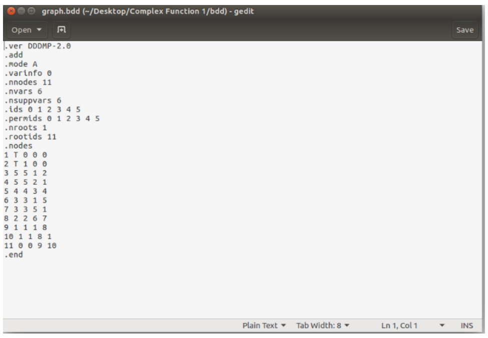

Figure 13: ComplexFunction1.bdd

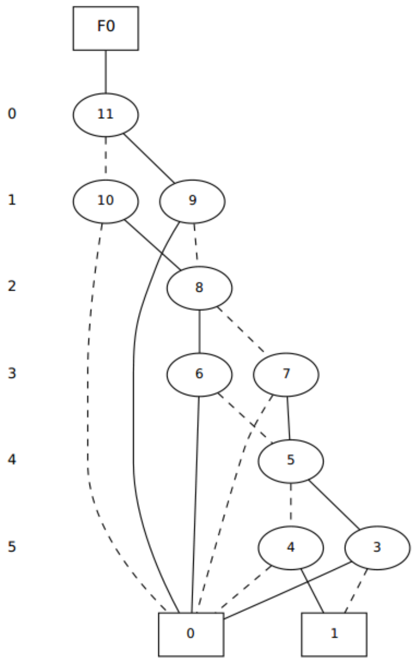

Figure 14: ComplexFunction1.dot

4.2 Milestone II: Boolean operations on BDD without CUDD library

In this Milestone, we construct the binary tree to represent BDDs, then execute an OR operation between two BDDs. Afterward, we traverse the resulted tree using in order traversal method to test our logic. The main objective behind this milestone is to give a small example of constructing and traversing the binary tree, as a preparation to Milestone III, in which we would construct a bigger binary tree and traverse upon its nodes. In addition, we compare the results obtained out of this example with a self-drawn example in order to test its credibility.

- **In-order traversal results of the Algorithm resulted tree**

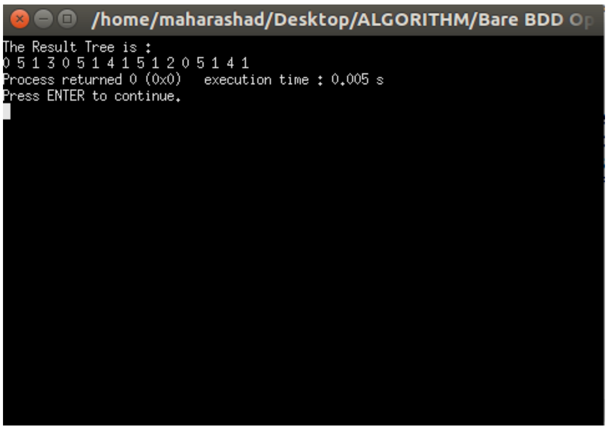

Figure 15: In-order traversal results

- **In-order traversal results of the self-Drawn resulted tree**

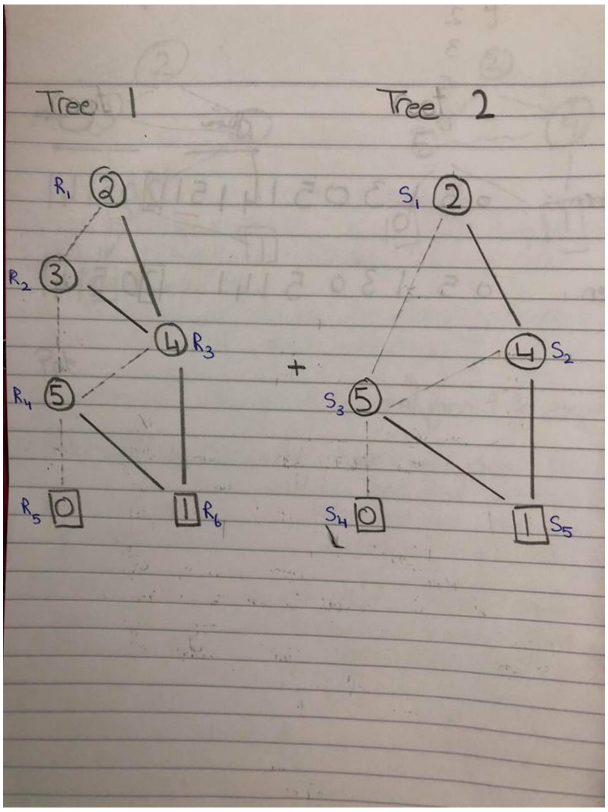

Figure 16: Or Operation

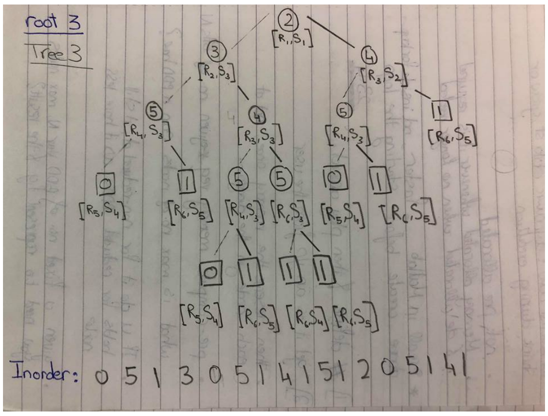

Figure 17: Result of OR operation

4.3 Milestone III: Understand .bdd files generated from SCOTS

SCOTS is an open source software tool for the synthesis of symbolic controllers for nonlinear control systems. The SCOTS takes a non-linear differential equation as an input and generates files in a certain format which is name.bdd format. The name.bdd files consist of two sections: The header and the list of nodes, where the header has a common (text) format, while the list of nodes is either in text or binary format. In our case, the list of nodes is in text format. These BDDs generated from SCOTS are actually in their Reduced Order Binary Decision Diagram (ROBDD).

Figure 18: SCOTS

4.3.1 Controller.bdd Text Files

They are files generated from SCOTS that represent the controller in form of binary decision diagram.

- Don't Cares

  As we mentioned before, these files are in the most reduced form which is called ROBDD. These ROBDDs are unlike prefect BDD, where every node points to two children right in the next level. However, the ROBDD representation does not follow this rule, a node points to another node that is in a level different from the next level. We observe from figure 16, basically they skip levels. Referring to Node-ID 4, which its dashed line skips a level and goes directly to the terminal node 1.

  In order to express the path to the terminal node from anywhere, we use bits zero and one, where zero represents the dashed line and one represents the solid line. For example, the path from Node-ID 4 to zero is 11, which means a solid line from Node-ID 4 to Node-ID 3, and another solid line from Node-ID 3 to terminal zero.

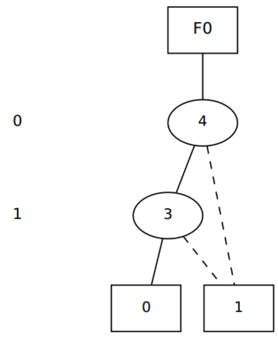

Figure 19: Nand.dot File

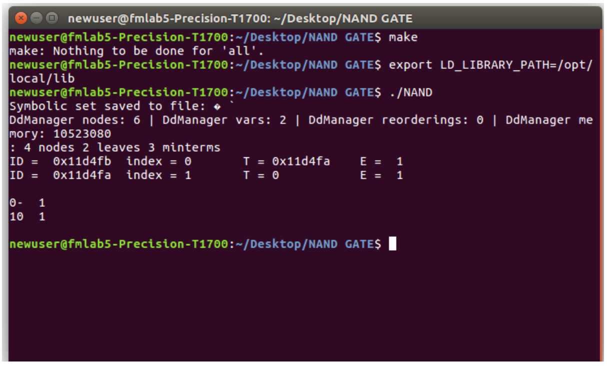

Figure 20: Action Bits output using CUDD library

4.4 Milestone IV: Implement BDD based controller in memory

After we have already understood the bdd files, now we should convert the content of the bdd files to a real data structure to be saved in memory. Thus, this step is split into two source codes.

1. Algorithm to generate hex memory of the BDD
2. Algorithm to traverse the BDD to identify the controller inputs

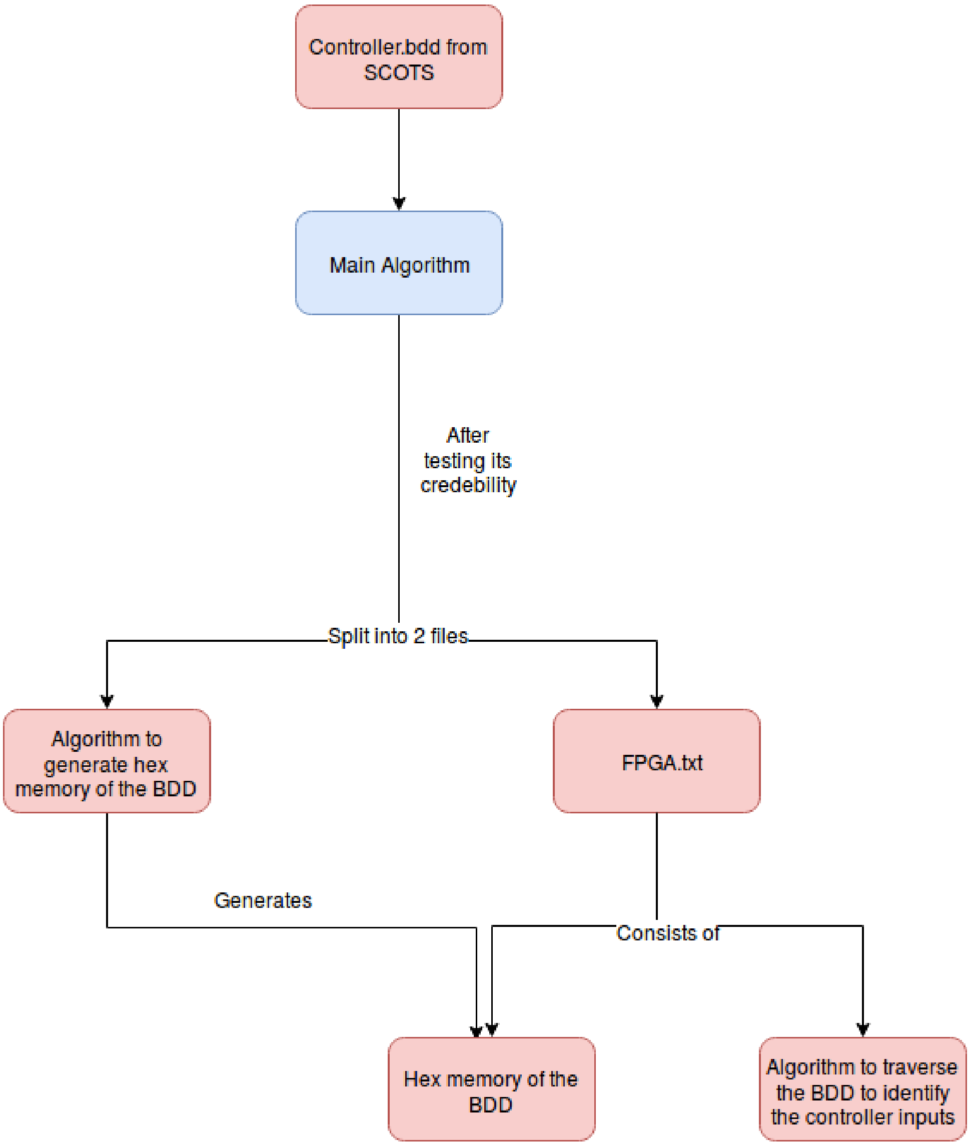

Figure 21: Algorithm sequence of events

4.4.1 Algorithm to generate hex memory data of the BDD

Here, we have constructed an algorithm that generates the memory data in hexadecimal format to be stored in the FPGA, we accomplish this through three important steps in the algorithm.

1. Extract data from controller.bdd file. As described in milestone I, how the controller.bdd are identified in terms of binary decision diagrams, and what does each column represent. The algorithm would read all these data and identify its meaning, as we have described before.

2. Convert it into a binary tree. After reading the controller.bdd file and identifying its meaning, we construct a whole binary tree based on these information.

3. Store the memory hex file. After constructing the full binary tree out of this controller.bdd file, we would save it as a hex format in a file called "FPGA.txt". This file contains the traversal algorithm we use to traverse the tree, which contains the memory data of the BDD, as well as the traversal algorithm to traverse this BDD.

- Algorithm to generate hex memory Flowchart [6]

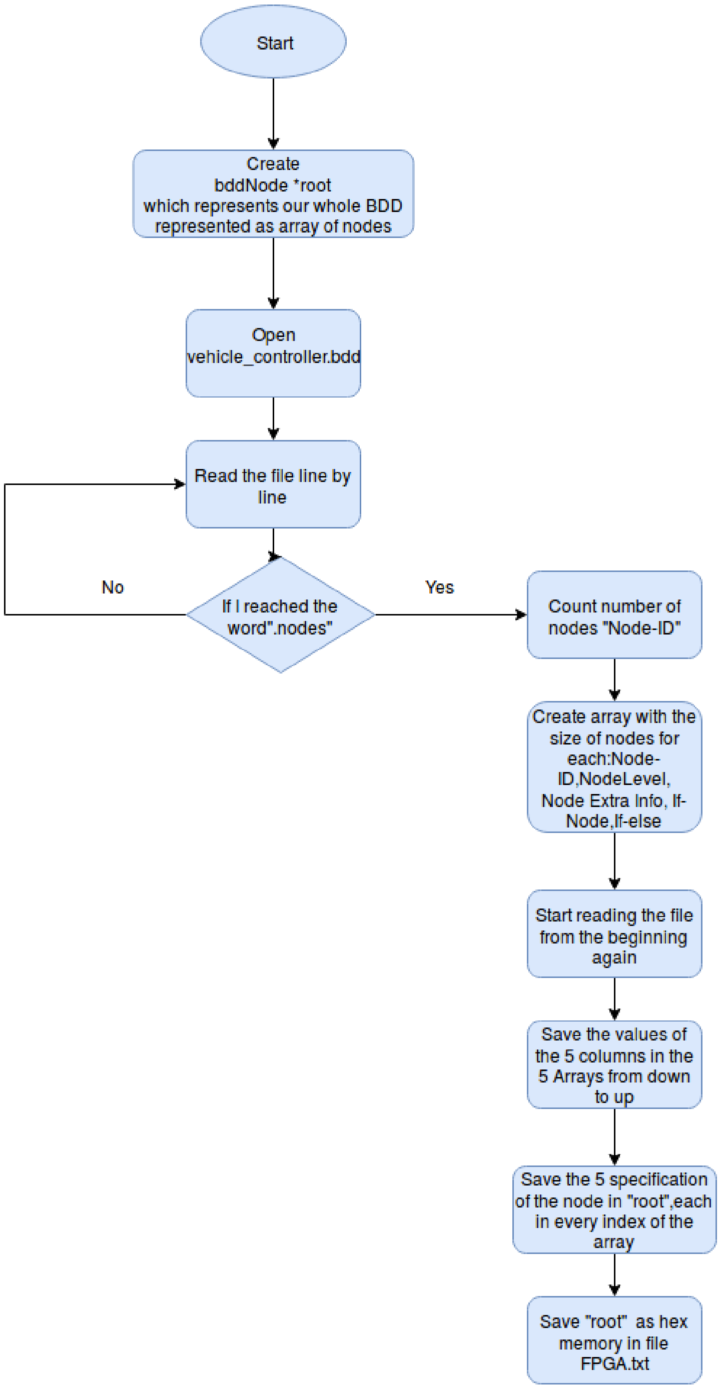

Figure 22: Algorithm to generate hex memory data Flow chart

4.4.2 Algorithm to traverse the BDD to identify the controller inputs

In this algorithm, we aim to identify the controller data, since this controller is a symbolic controller, the node represents the states and the inputs represent the path that leads to the terminal node. Since our main aim is to satisfy the controllers' requirements and reach terminal 1, we would need to traverse any given BDD to reach the terminal 1 in order to identify the needed controller inputs that would lead to terminal 1.

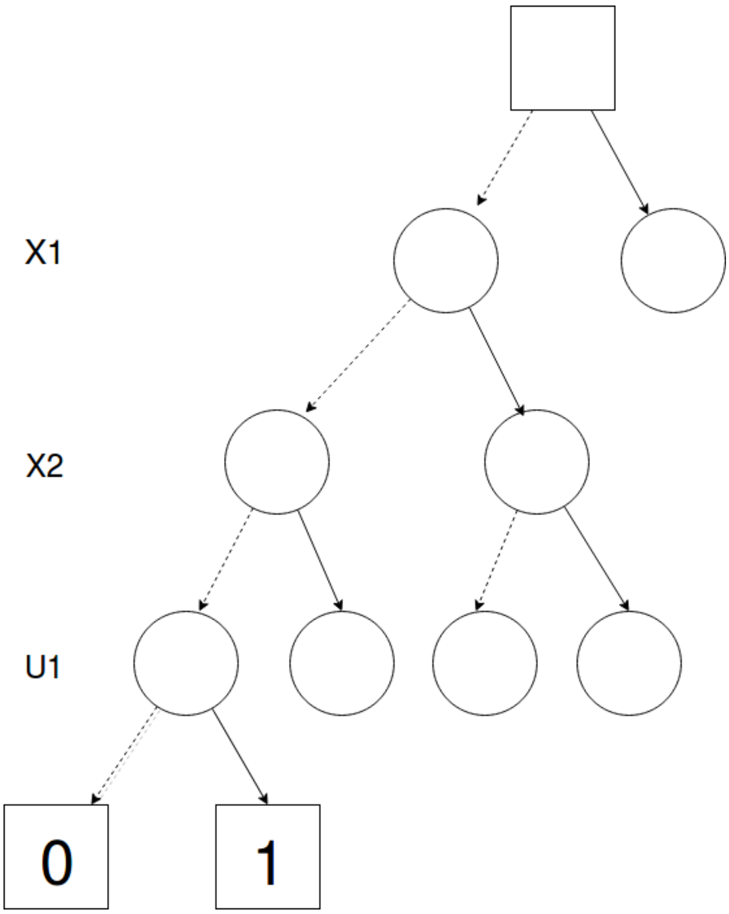

Figure 23: Binary Decision Diagram

After successfully constructing the whole binary tree that represents our controller BDD, saved it in the memory in file **FPGA.txt** using **Algorithm to generate hex memory of the BDD**, now we need to traverse the tree in order to identify the controller inputs.

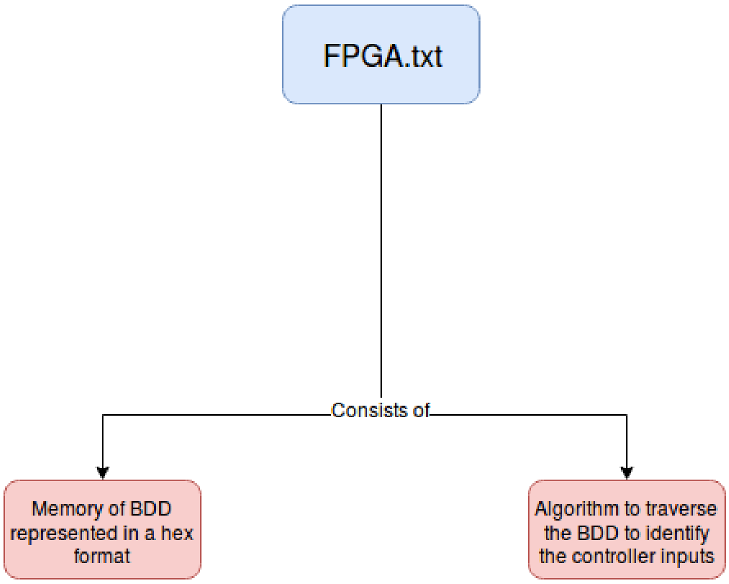

Figure 24: Generated FPGA.txt file contents

4.5 Limitations

In this section, we would discuss the limitations we have faced regarding mainly the controller.bdd file formats, dddmp package in Cudd library and how we overcame the problem.

- ADD format VS BDD format

Using CUDD Library, the controller.bdd files are generated in two formats, either in an ADD format or in BDD format.Most tools use the BDD format, such as SCOTS.

1. ADD format

   As we can observe in figure 25, it has two terminal nodes, Terminal 0 and Terminal 1.In addition, the path from a node to another is only represented through a solid line or a dashed line

```
.ver DDDMP-2.0
.add
.mode A
.varinfo 0
.nnodes 11
.nvars 6
.nsuppvars 6
.ids 0 1 2 3 4 5
.permids 0 1 2 3 4 5
.nroots 1
.rootids 11
.nodes
1 T 0 0 0
2 T 1 0 0
3 5 5 1 2
4 5 5 2 1
5 4 4 3 4
6 3 3 1 5
7 3 3 5 1
8 2 2 6 7
9 1 1 1 8
10 1 1 8 1
11 0 0 9 10
.end
```

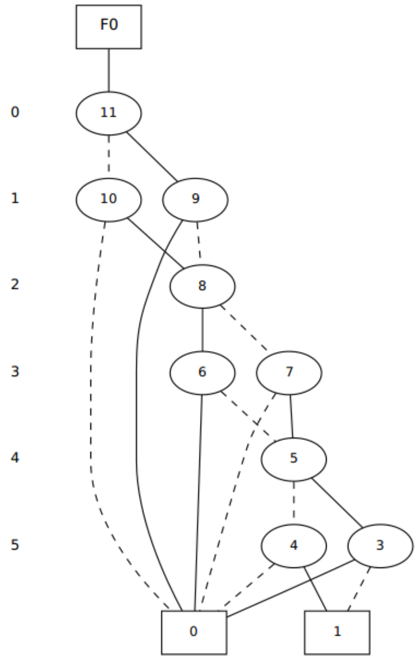

Figure 26: Binary decision diagram

Figure 25: ADD format files

2. BDD format

As shown in figure 27, there are nodes with negative values, these negative values are complemented edges, it means that when we reach these nodes, all paths afterward are complemented. Looking at figure 16, we observe that the diagram does not only have solid edges and dashed edges which means one and zero, but it has an additional line which is the complement line. As well as only having one terminal node called terminal 1. As a result, this format (BDD) makes our understanding of the decision diagram very hard and more complex.

```
.ver DDDMP-2.0
.mode A
.varinfo 0
.nnodes 9
.nvars 6
.nsuppvars 6
.ids 0 1 2 3 4 5
.permids 0 1 2 3 4 5
.nroots 1
.rootids -9
.nodes
1 T 1 0 0
2 5 5 1 -1
3 4 4 2 -2
4 3 3 1 3
5 3 3 3 1
6 2 2 4 5
7 1 1 1 6
8 1 1 6 1
9 0 0 7 8
.end
```

Figure 27: BDD format files

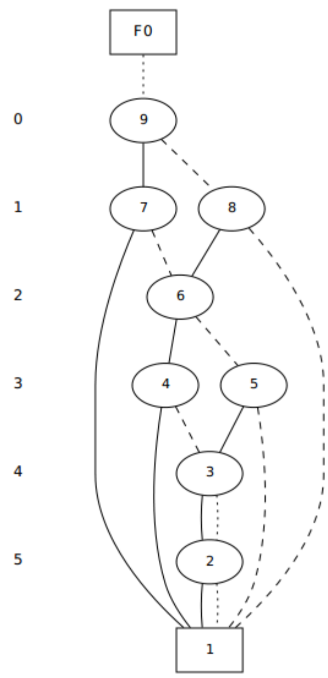

Figure 28: Binary decision diagram

Solution
In order to avoid complexity, we have chosen to use the ADD format files instead of BDD ones as it is more clear.Since most tools use BDD format as scots , we made a slight change in the source code of SCOTS using these methods

- Dddmp_cuddAddStore() ;
- Cudd_BddToAdd();

Steps
1.Open SCOTS
2.Open SymbolicSet.hh inside bdd file
3.Go to writeToFile() method.

4. Replace these lines

```
int storeReturnValue = Dddmp_cuddBddStore(
    mdest.getManager(),
    NULL,
    tosave.getNode(),
    //(char**)varnameschar, // char ** varnames, IN: array of variable names (or NULL)
    NULL, // char ** varnames, IN: array of variable names (or NULL)
    NULL,
    DDDMP_MODE_TEXT,
    // DDDMP_VARNAMES,
    DDDMP_VARIDS,
    NULL,
    file
  );
```

with

```
DddNode *bdd;
bdd=Cudd_BddToAdd(mdest.getManager(),bdd);
    int storeReturnValue = Dddmp_cuddAddStore(
          mdest.getManager(),
      NULL,
      bdd,
      //(char**)varnameschar, // char ** varnames, IN: array of variable names (or NULL)
      NULL, // char ** varnames, IN: array of variable names (or NULL)
      NULL,
      DDDMP_MODE_TEXT,
      // DDDMP_VARNAMES,
      DDDMP_VARIDS,
      NULL,
      file
    );
```

- Binary Format VS Text Format

SCOTS generate name.bdd files in a binary format so , we had to change this format to text inorder to be able to read the files .

Steps
1.Open SCOTS
2.Open SymbolicSet.hh inside bdd file
3.Go to writeToFile() method.
4.In cudd—BddStore() method,change DDDMP—MODE—BINARY to DDDMP—MODE—TEXT

5 Results and discussion

In this section, we show the output of our algorithms along with different examples.

5.1 Algorithm to traverse the BDD to identify the controller inputs

Here, we show the results of the algorithm used to traverse the BDD in order to generate all the possible input path that leads to terminal one in terms of zeros and ones which actually represent dash lines and solid lines. We would compare the automatically generated output when we compile the And gate algorithm using CUDD library we discussed in Milestone 1 with the **Algorithm to traverse the BDD to identify the controller inputs**.**Action bits** describes the path needed to reach terminal 1, for example, bit 1 means" go with the solid line", bit 0 means "then go with the dashed line" and so on till it reaches Terminal 1.We must also state that we have done this test before we split the algorithm into two algorithms as stated before.

1. **And Example**
   If we traced the BDD figure 32 starting from Node-ID 4, we would find that there is only one way to reach terminal 1 which is from Node-ID 4 to Node-Id 3 through a solid line, then from Node-ID 3 to Terminal 1 through a solid line, which means Action bits should be equal to 11.

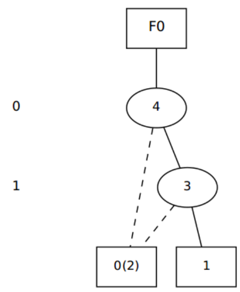

Figure 29: And.dot File

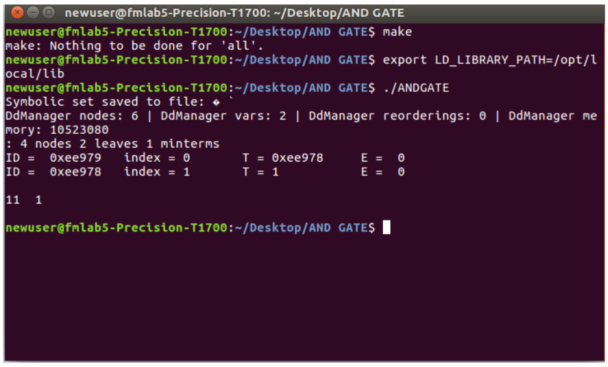

Figure 30: Action Bits output using CUDD library

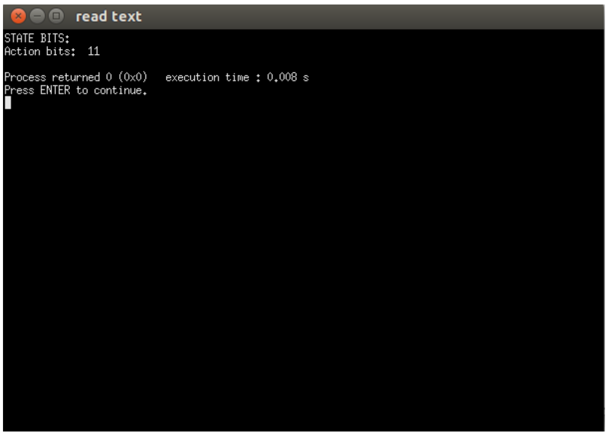

Figure 31: Action Bits output using our proposed Algorithm

2. **Nand Example**

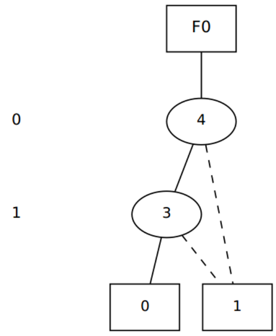

Figure 32: And.dot File

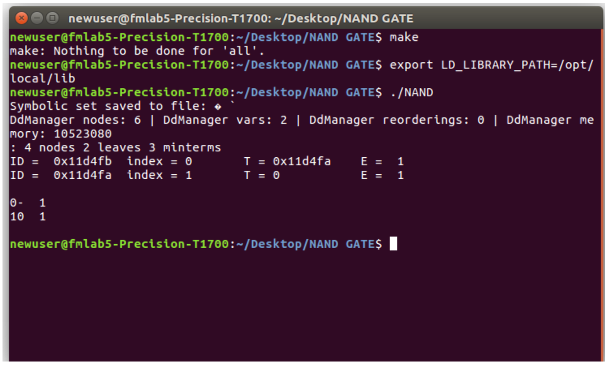

Figure 33: Action Bits output using CUDD library

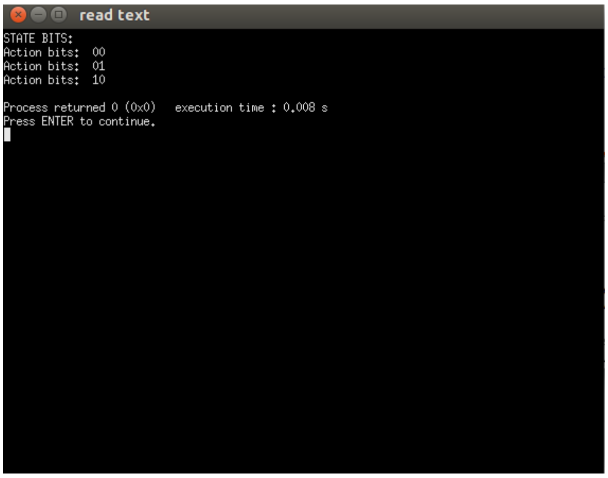

Figure 34: Action Bits output using our proposed Algorithm

3. **Complex Function 1 Example**

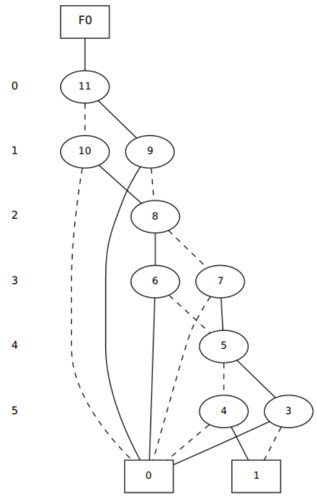

Figure 35: Complex1.dot File

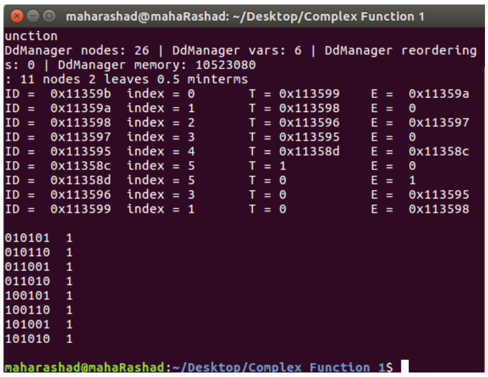

Figure 36: Action Bits output using CUDD library

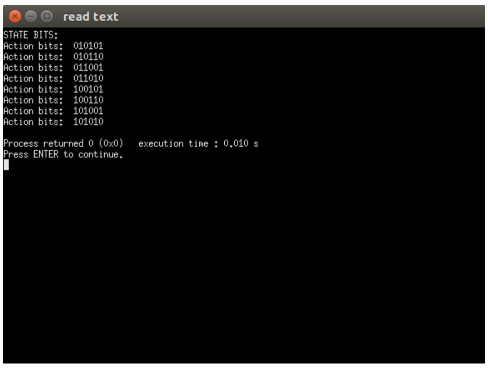

Figure 37: Action Bits output using our proposed Algorithm

5.2 Effect of our Algorithm in BDD2implement

As mentioned before, BDD2Implement is a C++ tool to generate hardware/software implementations of BDD-based symbolic controllers. Having the tools SCOTS that generate BDD-based symbolic controllers of general nonlinear dynamical systems, BDD2Implement takes the controller.bdd file from SCOTS and convert the BDD to truth table so the controller is represented as a boolean function. In this thesis, we have taken the controller.bdd file from SCOTS and converted the BDD to a data structure, put it in a file that traverses the tree to be saved in memory directly. Representing a BDD in a truth table in terms of space is inefficient. For example, having a boolean function with 100 variables would need (2^100) lines in order to represent all possible combination. On the other hand, representing the BDD using a data structure as binary trees with the same nature as BDD makes it more efficient, since the controller.bdd file generated from scots is already in the reduced, most simplified form of the BDD [ROBDD]

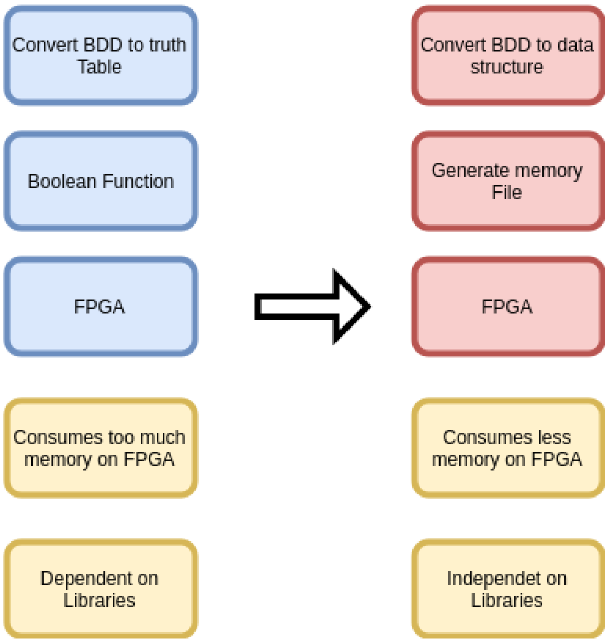

Figure 38: Effect of our Algorithm in BDD2IMPLEMENT

5.3 Conclusion

In this thesis, we are concerned about BDDs, how to understand and construct them. We propose a new method of representing symbolic controllers represented as BDDs on FPGAs or microcontrollers in general. We constructed a C++ implementation of BDDs that can read name.bdd files generated from SCOTS or any other tool. First, we read these files, understand them and then construct a binary tree to be saved as a hex format.In addition, we traverse the tree in order to extract all the right outputs of the controller. Afterwards, a template file is ready to be put in OpenCL then to microcontrollers.

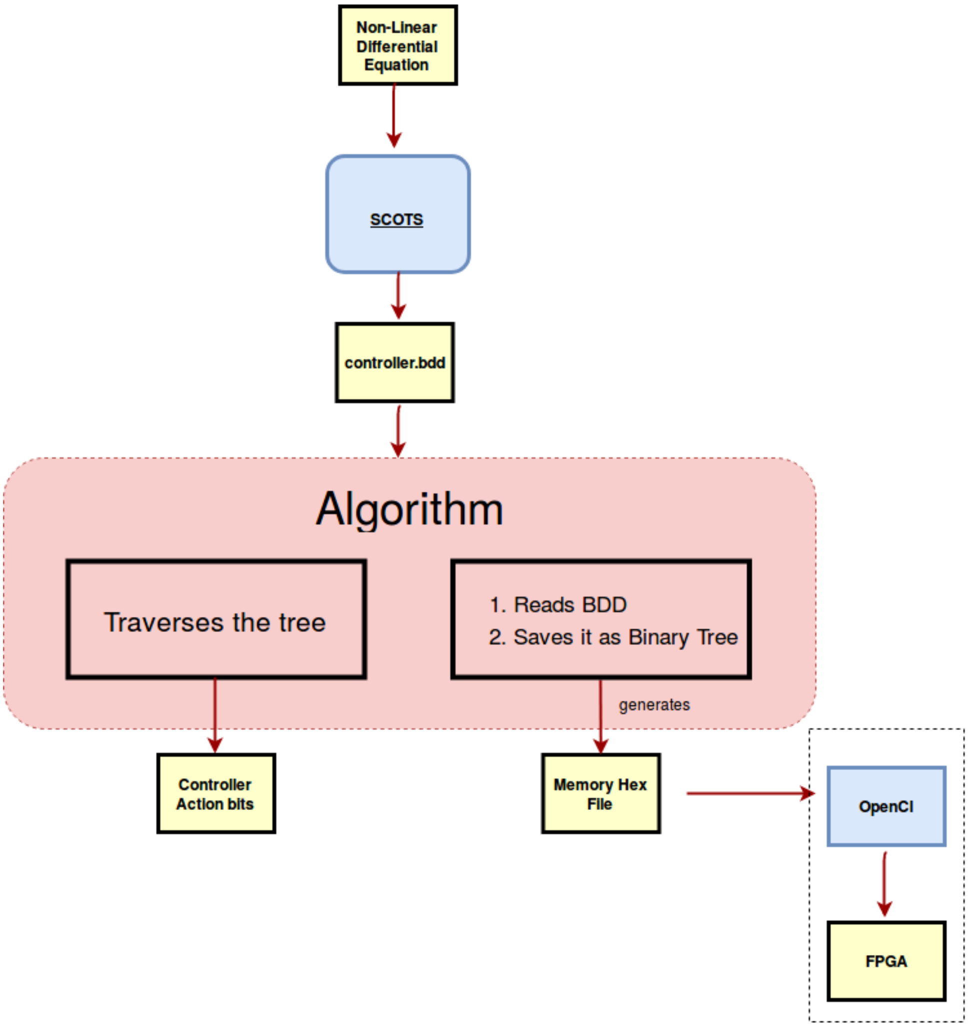

Figure 39: Sequence of events

6 Future Work

- Test the code on FPGA

  After we have successfully generated the memory of the BDD and the ”FPGA.txt” file is ready to be put directly in OpenCL, then Xilinx then tests it on FPGA. We can do this using several examples that have been generated already using scots such as vehicle 1, vehicle 2, unicycle or dcdc.

- Add the algorithm to BDD2implement

List of Figures

List of Algorithms

References

[1] BDD2implement . https://gitlab.lrz.de/hcs/BDD2Implement.

[2] Binary Decision Diagrams . https://www.slideshare.net/haroonrashidlone/binary-decision-diagrams.

[3] Construct a complete binary tree from given array in level order fashion . https://www.geeksforgeeks.org/construct-complete-binary-tree-given-array/.

[4] Construct Binary Tree from given Parent Array representation . https://www.geeksforgeeks.org/construct-a-binary-tree-from-parent-array-representation/.

[5] CUDD Tutorials . http://davidkebo.com/cudd.

[6] Given a binary tree, print all root-to-leaf paths . https://www.geeksforgeeks.org/given-a-binary-tree-print-all-root-to-leaf-paths/.

[7] Marco Benedetti. Istituto per la ricerca scientifica e tecnologica (irst) via sommarive 18, 38055 povo, trento, italy. 2005.

[8] Giordano Pola, Antoine Girard, and Paulo Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(10):2508–2516, 2008.

[9] Giordano Pola and Paulo Tabuada. Symbolic models for nonlinear control systems: Alternating approximate bisimulations. *SIAM Journal on Control and Optimization*, 48(2):719–733, 2009.

[10] Gunther Reissig and Matthias Rungger. Abstraction-based solution of optimal stopping problems under uncertainty. In *CDC*, pages 3190–3196, 2013.

[11] Gunther Reissig, Alexander Weber, and Matthias Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Transactions on Automatic Control*, 62(4):1781–1796, 2017.

[12] Matthias Rungger and Majid Zamani. Scots: A tool for the synthesis of symbolic controllers. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 99–104. ACM, 2016.

[13] Fabio Somenzi. Binary decision diagrams. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:303–368, 1999.

[14] Fabio Somenzi. Cudd: Cu decision diagram package-release 2.4. 0. *University of Colorado at Boulder*, 2009.

[15] Paulo Tabuada. Symbolic models for control systems. *Acta Informatica*, 43(7):477–500, 2007.

[16] Paulo Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

Appendix

1. And Gate

Algorithm 1: And Gate

```
#include <array>
#include <iostream>
#include "cuddObj.hh"
#include "util.h"
#include "dddmp.h"

void print_dd (DdManager *gbm, DdNode *dd, int n, int pr )
{
    printf("DdManager nodes: %ld | ", Cudd_ReadNodeCount(gbm)); /*Reports the number of live nodes in BDDs and ADDs*/
    printf("DdManager vars: %d | ", Cudd_ReadSize(gbm) ); /*Returns the number of BDD variables in existence*/
    printf("DdManager reorderings: %d | ", Cudd_ReadReorderings(gbm) ); /*Returns the number of times reordering has occurred*/
    printf("DdManager memory: %ld \n", Cudd_ReadMemoryInUse(gbm) ); /*Returns the memory in use by the manager measured in bytes*/
    Cudd_PrintDebug(gbm, dd, n, pr); // Prints to the standard output a DD and its statistics: number of nodes, number of leaves, number of minterms.
}

/**
 * Writes a dot file representing the argument DDs
 * @param the node object
 */
void write_dd (DdManager *gbm, DdNode *dd, char* filename)
{
    FILE *outfile; // output file pointer for .dot file
    outfile = fopen(filename,"w");
    DdNode **ddnodearray = (DdNode**)malloc(sizeof(DdNode*)); // initialize the function array
    ddnodearray[0] = dd;
    Cudd_DumpDot(gbm, 1, ddnodearray, NULL, NULL, outfile); // dump the function to .dot file
    free(ddnodearray);
    fclose (outfile); // close the file */
}
int main() {
    char filename[30];
    DdManager *gbm; /* Global BDD manager. */
    gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0); /* Initialize a new BDD manager. */
    DdNode *bdd, *x1, *x2;
    x1 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x2 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x2*/
    bdd = Cudd_bddAnd(gbm, x1, x2); /*Perform AND Boolean operation*/
    Cudd_Ref(bdd);
    FILE *file = fopen ("./FilesGenerated/graph.bdd","w");
   int storeReturnValue = Dddmp_cuddBddStore(
     gbm,
      NULL,
      bdd,
      //(char**)varnameschar, // char ** varnames, IN: array of variable names (or NULL)
      NULL, // char ** varnames, IN: array of variable names (or NULL)
      NULL,
      DDDMP_MODE_TEXT,
      // DDDMP_VARNAMES,
      DDDMP_VARIDS,
```

```
          NULL,
          file
        );


fclose(file);
if (storeReturnValue!=DDDMP_SUCCESS)
          throw "Error: Unable to write BDD to file.";
        else
          std::cout << "Symbolic set saved to file: "<< filename << std::endl;

                  /*Update the reference count for the node just created.*/
        bdd = Cudd_BddToAdd(gbm, bdd); /*Convert BDD to ADD for display purpose*/
        print_dd (gbm, bdd, 2,4);    /*Print the dd to standard output*/
        sprintf(filename, "./FilesGenerated/graph.dot"); /*Write .dot filename to a string
            */
        write_dd(gbm, bdd, filename);  /*Write the resulting cascade dd to a file*/

          Cudd_Quit(gbm);
}
```

2. Nand Gate

Algorithm 2: Nand Gate

```
#include <array>
#include <iostream>

#include "cuddObj.hh"
#include "util.h"
#include "dddmp.h"

void print_dd (DdManager *gbm, DdNode *dd, int n, int pr )
{
    printf("DdManager nodes: %ld | ", Cudd_ReadNodeCount(gbm)); /*Reports the number of live nodes in BDDs and ADDs*/
    printf("DdManager vars: %d | ", Cudd_ReadSize(gbm) ); /*Returns the number of BDD variables in existence*/
    printf("DdManager reorderings: %d | ", Cudd_ReadReorderings(gbm) ); /*Returns the number of times reordering has occurred*/
    printf("DdManager memory: %ld \n", Cudd_ReadMemoryInUse(gbm) ); /*Returns the memory in use by the manager measured in bytes*/
    Cudd_PrintDebug(gbm, dd, n, pr);  // Prints to the standard output a DD and its statistics: number of nodes, number of leaves, number of minterms.
}

/**
 * Writes a dot file representing the argument DDs
 * @param the node object
 */
void write_dd (DdManager *gbm, DdNode *dd, char* filename)
{
    FILE *outfile; // output file pointer for .dot file
    outfile = fopen(filename,"w");
    DdNode **ddnodearray = (DdNode**)malloc(sizeof(DdNode*)); // initialize the function array
    ddnodearray[0] = dd;
    Cudd_DumpDot(gbm, 1, ddnodearray, NULL, NULL, outfile); // dump the function to .dot file
    free(ddnodearray);
    fclose (outfile); // close the file */
}
int main() {
    char filename[30];
    DdManager *gbm; /* Global BDD manager. */
    gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0); /* Initialize a new BDD manager. */
    DdNode *bdd, *x1, *x2;
    x1 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x2 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x2*/
    bdd = Cudd_bddNand(gbm, x1, x2); /*Perform AND Boolean operation*/
    Cudd_Ref(bdd);
    FILE *file = fopen ("./bdd/graph.bdd","w");
  bdd = Cudd_BddToAdd(gbm, bdd);
    int storeReturnValue = Dddmp_cuddAddStore(
      gbm,
        NULL,
        bdd,
        //(char**varnameschar, // char ** varnames, IN: array of variable names (or NULL)
        NULL, // char ** varnames, IN: array of variable names (or NULL)
        NULL,
        DDDMP_MODE_TEXT,
        // DDDMP_VARNAMES,
        DDDMP_VARIDS,
        NULL,
        file
    );
```

```
fclose(file);
if (storeReturnValue!=DDDMP_SUCCESS)
    throw "Error: Unable to write BDD to file.";
  else
    std::cout << "Symbolic set saved to file: "<< filename << std::endl;

          /*Update the reference count for the node just created.*/
  /*Convert BDD to ADD for display purpose*/
  print_dd (gbm, bdd, 2,4);   /*Print the dd to standard output*/
  sprintf(filename, "./bdd/graph.dot"); /*Write .dot filename to a string*/
  write_dd(gbm, bdd, filename);  /*Write the resulting cascade dd to a file*/

    Cudd_Quit(gbm);
}
```

3. Complex Function 1

Algorithm 3: Complex Function 1

```

#include <array>
#include <iostream>

#include "cuddObj.hh"
#include "util.h"
#include "dddmp.h"
void print_dd (DdManager *gbm, DdNode *dd, int n, int pr )
{
    printf("DdManager nodes: %ld | ", Cudd_ReadNodeCount(gbm)); /*Reports the number of live nodes in BDDs and ADDs*/
    printf("DdManager vars: %d | ", Cudd_ReadSize(gbm) ); /*Returns the number of BDD variables in existence*/
    printf("DdManager reorderings: %d | ", Cudd_ReadReorderings(gbm) ); /*Returns the number of times reordering has occurred*/
    printf("DdManager memory: %ld \n", Cudd_ReadMemoryInUse(gbm) ); /*Returns the memory in use by the manager measured in bytes*/
    Cudd_PrintDebug(gbm, dd, n, pr);  // Prints to the standard output a DD and its statistics: number of nodes, number of leaves, number of minterms.
}

/**
 * Writes a dot file representing the argument DDs
 * @param the node object
 */
void write_dd (DdManager *gbm, DdNode *dd, char* filename)
{
    FILE *outfile; // output file pointer for .dot file
    outfile = fopen(filename,"w");
    DdNode **ddnodearray = (DdNode**)malloc(sizeof(DdNode*)); // initialize the function array
    ddnodearray[0] = dd;
    Cudd_DumpDot(gbm, 1, ddnodearray, NULL, NULL, outfile); // dump the function to .dot file
    free(ddnodearray);
    fclose (outfile); // close the file */
}
int main() { //x1 xor x2 * x3 xor x4 * x5 xor x6
 char filename[30];
    DdManager *gbm; /* Global BDD manager. */
    gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0); /* Initialize a new BDD manager. */
    DdNode *bdd, *x1, *x2, *bdd2, *x3, *x4 ,*bdd3,*x5,*x6, *x7,   *result;
    x1 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x2 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x2*/
    x3 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x4 = Cudd_bddNewVar(gbm);
    x5 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x6 = Cudd_bddNewVar(gbm);

    bdd = Cudd_bddXor(gbm, x1, x2); /*Perform OR Boolean operation*/
    bdd2=Cudd_bddXor(gbm, x3, x4);
    bdd3= Cudd_bddXor(gbm, x5, x6);
    x7=Cudd_bddAnd(gbm,bdd,bdd2);
    result=Cudd_bddAnd(gbm,x7,bdd3);



    Cudd_Ref(result);          /*Update the reference count for the node just created.*/
    result = Cudd_BddToAdd(gbm, result); /*Convert BDD to ADD for display purpose*/

```

```
  FILE *file = fopen ("./bdd/graph.bdd","w");
    int storeReturnValue = Dddmp_cuddAddStore(
     gbm,
      NULL,
     result,
      //(char**)varnameschar, // char ** varnames, IN: array of variable names (or NULL)
      NULL, // char ** varnames, IN: array of variable names (or NULL)
      NULL,
      DDDMP_MODE_TEXT,
      // DDDMP_VARNAMES,
      DDDMP_VARIDS,
      NULL,
      file
    );



    print_dd (gbm, result, 2,4);   /*Print the dd to standard output*/
    sprintf(filename, "./bdd/graph.dot"); /*Write .dot filename to a string*/
    write_dd(gbm, result, filename);  /*Write the resulting cascade dd to a file*/





    //fclose(file);

}
```

4. Complex Function 2

Algorithm 4: Complex Function 2

```
#include <array>
#include <iostream>

#include "cuddObj.hh"
#include "util.h"
#include "dddmp.h"

void print_dd (DdManager *gbm, DdNode *dd, int n, int pr )
{
    printf("DdManager nodes: %ld | ", Cudd_ReadNodeCount(gbm)); /*Reports the number of live nodes in BDDs and ADDs*/
    printf("DdManager vars: %d | ", Cudd_ReadSize(gbm) ); /*Returns the number of BDD variables in existence*/
    printf("DdManager reorderings: %d | ", Cudd_ReadReorderings(gbm) ); /*Returns the number of times reordering has occurred*/
    printf("DdManager memory: %ld \n", Cudd_ReadMemoryInUse(gbm) ); /*Returns the memory in use by the manager measured in bytes*/
    Cudd_PrintDebug(gbm, dd, n, pr); // Prints to the standard output a DD and its statistics: number of nodes, number of leaves, number of minterms.
}

/**
 * Writes a dot file representing the argument DDs
 * @param the node object
 */
void write_dd (DdManager *gbm, DdNode *dd, char* filename)
{
    FILE *outfile; // output file pointer for .dot file
    outfile = fopen(filename,"w");
    DdNode **ddnodearray = (DdNode**)malloc(sizeof(DdNode*)); // initialize the function array
    ddnodearray[0] = dd;
    Cudd_DumpDot(gbm, 1, ddnodearray, NULL, NULL, outfile); // dump the function to .dot file
    free(ddnodearray);
    fclose (outfile); // close the file */
}
int main() { //    X1.X2.X3.X4   +   X5.X6.X7.X8  +  X9.X10.X11.X12
    char filename[30];
    DdManager *gbm; /* Global BDD manager. */
    gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0); /* Initialize a new BDD manager. */
    DdNode *bdd,*bdd2,*bdd3,*bdd4, *x1, *x2,*x3, *x4,*x5, *x6,*x7, *x8,*x9, *x10,*x11, *x12,*And1,*And2,*And3,*And4, *And5,*And6,*And7, *And8,*And9, *Or1,*Or2,*Or3,*result;
    x1 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x2 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x2*/
    x3 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x4 = Cudd_bddNewVar(gbm);
    And1=Cudd_bddAnd(gbm, x1, x2);
    And2=Cudd_bddAnd(gbm, x3, x4);
    And3=Cudd_bddAnd(gbm, And1, And2); // X1.X2.X3.X4


    x5 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x6 = Cudd_bddNewVar(gbm);
    x7 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x8 = Cudd_bddNewVar(gbm);
    And4=Cudd_bddAnd(gbm, x5, x6);
    And5=Cudd_bddAnd(gbm, x7, x8);
    And6=Cudd_bddAnd(gbm, And4, And5); //X5.X6.X7.X8

```

```
    x9 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x10 = Cudd_bddNewVar(gbm);
    x11 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/
    x12 = Cudd_bddNewVar(gbm);
    And7=Cudd_bddAnd(gbm, x9, x10);
    And8=Cudd_bddAnd(gbm, x11, x12);
    And9=Cudd_bddAnd(gbm, And7, And8); //X9.X10.X11.X12


    Or1=Cudd_bddOr(gbm, And3, And6);
    result=Cudd_bddOr(gbm, Or1, And9);
    Cudd_Ref(result);
    FILE *file = fopen ("./bdd/graph.bdd","w");
    result = Cudd_BddToAdd(gbm, result); /*Convert BDD to ADD for display purpose*/
   int storeReturnValue = Dddmp_cuddAddStore(
     gbm,
      NULL,
      result,
      //(char**)varnameschar, // char ** varnames, IN: array of variable names (or NULL
          )
      NULL, // char ** varnames, IN: array of variable names (or NULL)
      NULL,
      DDDMP_MODE_TEXT,
      // DDDMP_VARNAMES,
      DDDMP_VARIDS,
      NULL,
      file
    );


fclose(file);
if (storeReturnValue!=DDDMP_SUCCESS)
      throw "Error: Unable to write BDD to file.";
    else
      std::cout << "Symbolic set saved to file: "<< filename << std::endl;

              /*Update the reference count for the node just created.*/

    print_dd (gbm, result, 2,4);   /*Print the dd to standard output*/
    sprintf(filename, "./bdd/graph.dot"); /*Write .dot filename to a string*/
    write_dd(gbm, result, filename);  /*Write the resulting cascade dd to a file*/








      Cudd_Quit(gbm);
}
```

5. OR Operation on two BDDs

Algorithm 5: OR Operation on two BDDs

```
#include<stdio.h>
#include<stdlib.h>

//struct node* node;
struct node
{
    int data;//data
    struct node *dashedline;//links
    struct node *solidline;
};

/* newNode() allocates a new node with the given data and NULL left and
   right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));// Allocate memory
        for new node
    node->data = data;// Assign data to this node
    node->dashedline = NULL; // Initialize left and right children as NULL
    node->solidline = NULL;
    return(node);


}
void inorder(struct node * node) // method to help output the result in inorder
     traversal
{
    if (!node)
        return;

    /* first recur on left child */
    inorder(node->dashedline);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    inorder(node->solidline);
}


struct node* OR(struct node *root1, struct node *root2 ) { //temps are integers while f
      and g are pointers

    int  val;
    struct node *newdashedF; //new pointers to be used based on node choices
    struct node *newsolidF;
    struct node *newdashedG;
    struct node *newsolidG;


    if(!root1 && !root2) { // if they are both null
        return NULL;
    }


    else if(!root2) { // ig root2 is NULL
        val=root1->data;
        return root1;
    }


```

```
    else if ( !root1) { //if root1 is null
        val=root2->data;
        return root2;
    }



    else if ((root1->data ==0 || root1->data==1) && (root2->data ==0 || root2->data==1)
        ) { // if both of them are terminal nodes then we apply or function
        val=root1->data | root2->data;
        newdashedF=root1->dashedline;
        newdashedG=root2->dashedline;
        newsolidF=root1->solidline;
        newsolidG=root2->solidline;
    }


    else if (root1->data==root2->data) {// if they are equal
        val=root1->data;
        newdashedF=root1->dashedline;
        newdashedG=root2->dashedline;
        newsolidF=root1->solidline;
        newsolidG=root2->solidline;
    }


    else if(root1->data <  root2->data ) {     // if root1 is smaller that root2
        if (root1->data==0 || root1->data ==1) { // if there is a terminal node and
            normal one, get the normal one
            val=root2->data;
            newdashedF=root1;
            newdashedG=root2->dashedline;
            newsolidF=root1;
            newsolidG=root2->solidline;

        }
        else {
            val=root1->data;
            newdashedF=root1->dashedline;
            newdashedG=root2;
            newsolidF=root1->solidline;
            newsolidG=root2;
        }
    }


    else  if(root1->data >  root2->data ) {
        if (root2->data==0 || root2->data ==1) { // if there is a terminal node and
            normal one, get the normal one
            val=root1->data;
            newdashedF=root1->dashedline;
            newdashedG=root2;
            newsolidF=root1->solidline;
            newsolidG=root2;

        }
        else {
            val=root2->data;
            newdashedF=root1;
            newdashedG=root2->dashedline;
            newsolidF=root1;
            newsolidG=root2->solidline;
        }
    }

    struct node *root3=newNode(val);
```

```
    root3->solidline=OR(newsolidF, newsolidG);
    root3->dashedline=OR(newdashedF , newdashedG);
    return root3;


}



int main()
{
    /* Let us construct below tree
                  2
                 / \
                3
                  \
                |   4
                  /

                5
                |    |

                0    1


              */
    struct node *root1  = newNode(2);
    root1->dashedline = newNode(3);
    root1->solidline =                             root1->dashedline->solidline=newNode(4);
    root1->dashedline->dashedline =                root1->solidline->dashedline= root1->dashedline->solidline->dashedline =newNode(5);
    root1->dashedline->dashedline->dashedline =       root1->solidline->dashedline->dashedline= root1->dashedline->solidline->dashedline->dashedline=  newNode(0);
    root1->solidline->solidline =                  root1->dashedline->dashedline->solidline=  root1->dashedline->solidline->solidline=   newNode(1);
    /* Let us construct below tree
                2
               / \
              5 -  4
              | \ |
              0  1

                */
    struct node *root2  = newNode(2);
    root2->solidline = newNode(4);
    root2->dashedline =                 root2->solidline->dashedline= newNode(5);
    root2->dashedline->dashedline =     root2->solidline->dashedline->dashedline= newNode(0);
    root2->solidline->solidline =       root2->dashedline->solidline= root2->solidline->dashedline->solidline=newNode(1);
    struct node *root3= OR(root1, root2);
    //int x=NULL;
    //printf("%d ", x);

//
    printf("The Result Tree is :\n");

    // printf("%d ", root3);
    inorder(root3);
    return 0;

}
```

6. Algorithm to generate hex memory of the BDD

Algorithm 6: Algorithm to generate hex memory of the BDD

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include "cuddObj.hh"
#include "CuddMintermIterator.hh"
#include "dddmp.h"

//#include <util.h>

#include <sstream>
#include <vector>
#include <cstdlib>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
#include <bits/stdc++.h>
/*BDD_INPUT_BITS represent number of bits the user enter for the states , if it is =2 , states[2]*/
#define BDD_INPUT_BITS 0
/*BDD_output_BITS represent number of bits of the input that this algorithm generates , which is the path for terminal 1*/
#define BDD_OUTPUT_BITS 24

#define arraySize 200000 // so i hve actionBIts={1010,1111,......} since I dont know all the path possibility that I have , i made an array of size 1000
using namespace std;
Cudd* ddmgr_;
size_t dim_;
/* var: eta_
   * dim_-dimensional vector containing the grid node distances */
double* eta_;
/* var: z_
   * dim_-dimensional vector containing the measurement error bound */
double* z_;
/* var: firstGridPoint_
   * dim_-dimensinal vector containing the real values of the first grid point */
double* firstGridPoint_;
/* var: firstGridPoint_
   * dim_-dimensinal vector containing the real values of the last grid point */
double* lastGridPoint_;
/* var: nofGridPoints_
   * integer array[dim_] containing the grid points in each dimension */
size_t* nofGridPoints_;
/* read the SymbolicSet information from file*/
/* var: nofBddVars_
   * integer array[dim_] containing the number of bdd variables in each dimension */
size_t* nofBddVars_;
/* var: indBddVars_
   * 2D integer array[dim_][nofBddVars_] containing the indices (=IDs) of the bdd variables */
size_t** indBddVars_;
/* var: nvars_
   * total number of bdd variables representing the support of the set */
size_t nvars_;
/* var: symbolicSet_
   * the bdd representing the set of points */
BDD symbolicSet_;
/* var: iterator_
   * class to iterate over all elements in the symbolic set*/
CuddMintermIterator* iterator_;
```

```

void print_dd(DdManager* gbm, DdNode* dd, int n, int pr)
{
    printf("DdManager nodes: %ld | ", Cudd_ReadNodeCount(gbm)); /*Reports the number of live nodes in BDDs and ADDs*/
    printf("DdManager vars: %d | ", Cudd_ReadSize(gbm)); /*Returns the number of BDD variables in existence*/
    printf("DdManager reorderings: %d | ", Cudd_ReadReorderings(gbm)); /*Returns the number of times reordering has occurred*/
    printf("DdManager memory: %ld \n", Cudd_ReadMemoryInUse(gbm)); /*Returns the memory in use by the manager measured in bytes*/
    Cudd_PrintDebug(gbm, dd, n, pr); // Prints to the standard output a DD and its statistics: number of nodes, number of leaves, number of minterms.
}
void readMembersFromFile(const char* filename)
{
    /* open file */
    std::ifstream bddfile(filename);
    if (!bddfile.good()) {
        std::ostringstream os;
        os << "Error: Unable to open file:" << filename << "'.";
        throw std::runtime_error(os.str().c_str());
    }
    /* read dimension from file */
    std::string line;
    while (!bddfile.eof()) {
        std::getline(bddfile, line);
        if (line.substr(0, 6) == "#scots") {
            if (line.find("dimension") != std::string::npos) {
                std::istringstream sline(line.substr(line.find(":") + 1));
                sline >> dim_;
            }
        }
    }
    if (dim_ == 0) {
        std::ostringstream os;
        os << "Error: Could not read dimension from file: " << filename << ". ";
        os << "Was " << filename << " created with scots::SymbolicSet::writeToFile?";
        throw std::runtime_error(os.str().c_str());
    }
    z_ = new double[dim_];
    eta_ = new double[dim_];
    lastGridPoint_ = new double[dim_];
    firstGridPoint_ = new double[dim_];
    nofGridPoints_ = new size_t[dim_];
    nofBddVars_ = new size_t[dim_];
    /* read eta/first/last/no of grid points/no of bdd vars */
    bddfile.clear();
    bddfile.seekg(0, std::ios::beg);
    int check = 0;
    while (!bddfile.eof()) {
        std::getline(bddfile, line);
        if (line.substr(0, 6) == "#scots") {
            /* read eta */
            if (line.find("eta") != std::string::npos) {
                check++;
                std::istringstream sline(line.substr(line.find(":") + 1));
                for (size_t i = 0; i < dim_; i++)
                    sline >> eta_[i];
            }
            /* read z */
            if (line.find("measurement") != std::string::npos) {
                check++;
                std::istringstream sline(line.substr(line.find(":") + 1));
                for (size_t i = 0; i < dim_; i++)
                    sline >> z_[i];
```

```
            }
            /* read first grid point*/
            if (line.find("first") != std::string::npos) {
                check++;
                std::istringstream sline(line.substr(line.find(":") + 1));
                for (size_t i = 0; i < dim_; i++)
                    sline >> firstGridPoint_[i];
            }
            /* read last grid point*/
            if (line.find("last") != std::string::npos) {
                check++;
                std::istringstream sline(line.substr(line.find(":") + 1));
                for (size_t i = 0; i < dim_; i++)
                    sline >> lastGridPoint_[i];
            }
            /* read no of grid points */
            if (line.find("number") != std::string::npos) {
                check++;
                std::istringstream sline(line.substr(line.find(":") + 1));
                for (size_t i = 0; i < dim_; i++) {
                    sline >> nofGridPoints_[i];
                    if (nofGridPoints_[i] == 1)
                        nofBddVars_[i] = 1;
                    else
                        nofBddVars_[i] = (size_t)std::ceil(log2(nofGridPoints_[i]));
                }
            }
        }
        if (check == 5)
            break;
    }
    if (check < 5) {
        std::ostringstream os;
        os << "Error: Could not read all parameters from file: " << filename << ". ";
        os << "Was " << filename << " created with scots::SymbolicSet::writeToFile?";
        throw std::runtime_error(os.str().c_str());
    }
    /* read index of bdd vars  */
    indBddVars_ = new size_t*[dim_];
    bddfile.clear();
    bddfile.seekg(0, std::ios::beg);
    check = 0;
    while (!bddfile.eof()) {
        std::getline(bddfile, line);
        if (line.substr(0, 6) == "#scots") {
            if (line.find("index") != std::string::npos) {
                check++;
                std::istringstream sline(line.substr(line.find(":") + 1));
                for (size_t i = 0; i < dim_; i++) {
                    indBddVars_[i] = new size_t[nofBddVars_[i]];
                    for (size_t j = 0; j < nofBddVars_[i]; j++)
                        sline >> indBddVars_[i][j];
                }
            }
        }
    }
    if (check != 1) {
        std::ostringstream os;
        os << "Error: Could not read bdd indices from file: " << filename << ".";
        os << "Was " << filename << " created with scots::SymbolicSet::writeToFile?";
        throw std::runtime_error(os.str().c_str());
    } /* close file */
    bddfile.close();
    /* number of total variables */
    nvars_ = 0;
    for (size_t i = 0; i < dim_; i++)
```

```
        for (size_t j = 0; j < nofBddVars_[i]; j++)
            nvars_++;
}
typedef struct node {
    /*nodeID*/
    int data;
    /*represent level of the node*/
    int levelnum;
    /*the var internal index which is the 3rd column in the .bdd file .. for the terminals use*/
    int index;

    /*Index in the root array of the dashed line node of a node not values*/
    int dashedline;
    /*Index in the root array of the dashed line node of a node not values*/
    int solidline;

} bddNode;

/*The binary tree constructed out of the BDD*/
bddNode* root;

/*  In this main method the .bdd file name should be changed everytime you test a new file
    we read values name.bdd files , count number of nodes , construct an array of nodes
    inwhich every index contains the five specification of a node which are data, levelnum,
    index ,solidline,dashedline.After constructing a tree out of the .bdd file ,we save
    the tree as a hex data in a new file "FPGA.txt" which has the traversal algorithm that
    traverses the tree.

    This method makes array of nodes root[],it gets the count variable from main
    method ,count variable is the number of parent nodes,
    for example if count =27 , makes 27 nodes and put them in the array of nodes,these are the parent nodes->ARRNodeId[],the first column of the file.
    Afterwards it checks each line , checking solid line and dashed lines,connecting the parent with the corresponding solid line and dashed lines from within the node array

 */

int main()
{
    Cudd ddmgr;
    int newID = 0;
    ddmgr_ = &ddmgr;
    const char* filename = "vehicle_controller.bdd";
    iterator_ = NULL;
    /* read the SymbolicSet members from file */
    readMembersFromFile(filename);
    int* composeids = NULL;
    Dddmp_VarMatchType match = DDDMP_VAR_MATCHIDS;
    /* do we need to create new variables ? */
    if (newID) {
        /* we have to create new variable id's and load the bdd with those new ids */
        match = DDDMP_VAR_COMPOSEIDS;
        /* allocate memory for comopsids */
        size_t maxoldid = 0;
        for (size_t i = 0; i < dim_; i++)
            for (size_t j = 0; j < nofBddVars_[i]; j++)
                maxoldid = ((maxoldid < indBddVars_[i][j]) ? indBddVars_[i][j] : maxoldid);
        composeids = new int[maxoldid + 1];
        /* match old id's (read from file) with newly created ones */
        for (size_t i = 0; i < dim_; i++) {
```

```
            for (size_t j = 0; j < nofBddVars_[i]; j++) {
                BDD bdd = ddmgr.bddVar();
                composeids[indBddVars_[i][j]] = bdd.NodeReadIndex();
                indBddVars_[i][j] = bdd.NodeReadIndex();
            }
        }
        /* number of total variables */
    }
    /* load bdd */
    FILE* file1 = fopen(filename, "r");
    if (file1 == NULL) {
        std::ostringstream os;
        os << "Error: Unable to open file:" << filename << "'.";
        throw std::runtime_error(os.str().c_str());
    }
    DdNode* bdd = Dddmp_cuddBddLoad(ddmgr.getManager(),
        match,
        NULL,
        NULL,
        composeids,
        DDDMP_MODE_TEXT,
        NULL,
        file1);
    fclose(file1);

    BDD tmp(ddmgr, bdd);
    //symbolicSet_=tmp;// bdd object // segmentationfault

    DdNode* result;
    //cudd mgr;
    bdd = Cudd_BddToAdd(ddmgr.getManager(), bdd);
    FILE* file2 = fopen("./bdd/graph.bdd", "w");
    Dddmp_cuddAddStore(
        ddmgr.getManager(),
        NULL,
        bdd,
        //(char**)varnameschar, // char ** varnames, IN: array of variable names (or
            NULL)
        NULL, // char ** varnames, IN: array of variable names (or NULL)
        NULL,
        DDDMP_MODE_TEXT,
        // DDDMP_VARNAMES,
        DDDMP_VARIDS,
        NULL,
        file2);
    print_dd(ddmgr.getManager(), result, 2, 4);
    delete[] composeids;

    int rootindex;
    int solidindex;
    int levelindex;
    int dashedindex;
    int internalindex;
    int i;
    char actionBits[arraySize];
    int stateBits[BDD_INPUT_BITS];

    int valRoot;
    int valSolid;
    int valDashed;
    /* number of nodes in a BDD*/
    int NumOfNodes = 0;
    ifstream inFile;
    /* Variable string used to loop on the header of .bdd file*/
    string candidate;
    /* The word ".nodes" is the last word in the .bdd file before the BDD data starts
```

```
        */
    string item = ".nodes";
    /*represent each column of the .bdd file */
    string nodeID, levelID, internalID, ifID, elseID;
    /*Open the .bdd file just to count number of nodes available in this bdd*/
    inFile.open("./bdd/graph.bdd");

    /*if (inFile.fail()) {
        cerr << "Hard Luck ! error opening it ! :D " << endl;
    }*/
    /*loop on the string which is mainly the header data*/
    while (inFile >> candidate) {
        /* if the word ".nodes " is found */
        if (item == candidate) {
            /*loop to count NumOfNodes available in a .bdd file*/
            for (i = 0; inFile >> nodeID >> levelID >> internalID >> ifID >> elseID; i++) { // as this is going on , go on
                NumOfNodes++; //number of nodes
            }
        }
    }

    inFile.close();
    /*construct an array for each column data with size of NUmOfNodes*/

    string ARRnodeID[NumOfNodes], ARRlevelID[NumOfNodes], ARRInternalID[NumOfNodes], ARRifID[NumOfNodes], ARRelseID[NumOfNodes];
    bddNode temp;
    /*Determine the size of our tree which is size should be only the number of nodes */
    root = (bddNode*)malloc(sizeof(bddNode) * NumOfNodes);
    int rootArray[NumOfNodes];
    /*These are number of nodes needed later for decrementing*/
    int count1 = NumOfNodes;
    int count2 = NumOfNodes;

    /*Open the .bdd file again to read the bdd values*/
    inFile.open("./bdd/graph.bdd");
    while (inFile >> candidate) {
        if (item == candidate) {
            for (int i = 0; inFile >> nodeID >> levelID >> internalID >> ifID >> elseID; i++) {

                ARRnodeID[i] = nodeID;
                ARRlevelID[i] = levelID;
                ARRInternalID[i] = internalID;
                ARRifID[i] = ifID;
                ARRelseID[i] = elseID;

                stringstream mainRoot(ARRnodeID[i]); // construct root
                stringstream level(ARRlevelID[i]);
                stringstream level2(ARRInternalID[i]);
                stringstream solid(ARRifID[i]); // construct solid line of root
                stringstream dashed(ARRelseID[i]); //construct dashed line of root
                mainRoot >> valRoot; //
                level >> levelindex; //the level i am in
                level2 >> internalindex; //internal index is the third row which is mostly important for the terminal values
                solid >> valSolid; // change string solid to int valsolid
                dashed >> valDashed;

                if (i == 0) {
                    /* Start inserting data in the root[] in reverse {11,12,,,,1}*/
                    for (int w = 0; count1 > 0; w++) {
                        /*rootArray[] is another array same as our main root[] , we would use it to manipulate data*/
```

```
                        rootArray[w] = count1;

                        root[w].data = count1;
                        count1--;
                    }
                }
                /*Loop in rootArray[] to know the index of the solidline node of the present node*/
                for (int q = 0; q < NumOfNodes; q++) {
                    /*If we found the solidline node , set the solidindex to be the index of the array*/
                    if (rootArray[q] == valSolid) {
                        solidindex = q;
                        break;
                    }
                }

                for (int q = 0; q < NumOfNodes; q++) {

                    if (rootArray[q] == valDashed) {
                        dashedindex = q;
                        break;
                    }
                }
                /*start by reverse so root[]={14,13,12,11,10.....1}*/
                if (count2 > 0) {
                    root[count2 - 1].solidline = solidindex; // did an array with all the main nodes , then here connecting them with their corresponding solid and dashed
                    root[count2 - 1].dashedline = dashedindex; //starting by reverse // since it is flipped
                    root[count2 - 1].levelnum = levelindex;
                    root[count2 - 1].index = internalindex;
                    count2--;
                }
            }
        }
    }

    /*get the maximum level of the BDD which is the level of the last node before the terminal*/
    int maxlevel = root[NumOfNodes - 3].levelnum;
    /*make the teminal nodes belong to a level number , which is the maxlevel+1*/
    int terminalLevel = maxlevel + 1;
    root[NumOfNodes - 2].levelnum = terminalLevel; // these are terminal  nodes , setting their level number= maximum level +1
    root[NumOfNodes - 1].levelnum = terminalLevel;

    root[NumOfNodes - 2].solidline = NULL; // these are terminal  nodes , setting their level number= maximum level +1
    root[NumOfNodes - 2].dashedline = NULL;

    root[NumOfNodes - 1].solidline = NULL; // these are terminal  nodes , setting their level number= maximum level +1
    root[NumOfNodes - 1].dashedline = NULL;

    size_t size = 0;
    /*User enters the StateBits with size of BDD_INPUT_BITS*/
    while (size < BDD_INPUT_BITS) {
        cin >> stateBits[size];
        size++;
    }
    /*Start the traversal*/
    //getControlAction(stateBits, actionBits, terminalLevel);

    /*here we export root data which are the data,levelnum,index,dashedline,solidline
```

```
    as a byte array to a text file
*each node has these 5 integers, where each integer is 4 bytes , so each node size
    is 20 bytes
so the total number of bytes= total number of nodes * 20*/

std::fstream file;

file.open("FPGA.txt", std::fstream::in | std::fstream::out | std::fstream::app);

file << "char* controllerData = {";
char* bytePtr = (char*)&root[0];

unsigned int numBytes = NumOfNodes * sizeof(bddNode);

for (unsigned int i = 0; i < numBytes; i++) {

    file << "0x";

    file << std::hex << std::uppercase << static_cast<unsigned int>(*bytePtr);

    bytePtr++;

    if (i != (numBytes - 1))

        file << ",";
}

file << "};";

file.close();

/*In this section we copy the traversal code into the file that contains the BDD
    hex data which is data.txt

*/

fstream files;

// Input stream class to
// operate on files.

/*This is to copy the traversal code from TraversalCode.txt to the file that has
    the BDD hex data which is FPGA.txt */
ifstream ifile("TraversalCode.txt", ios::in);

// Output stream class to
// operate on files.
ofstream ofile("FPGA.txt", ios::out | ios::app);

// check if file exists
if (!ifile.is_open()) {

    // file not found (i.e, not opened).
    // Print an error message.
    cout << "file not found";
}
else {
    // then add more lines to
    // the file if need be
    ofile << ifile.rdbuf();
}
string word;

// opening file
file.open("FPGA.txt");

```

```
    // extracting words form the file
    while (file >> word) {
    }

    return 0;
}
```

7. FPGA.txt , Algorithm to traverse the BDD to identify the controller inputs

Algorithm 7: FPGA.txt

```
char* controllerData = {0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B,0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F,0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D,0x6E,0x6F,0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,0xFFFFFF80,0xFFFFFF81,0xFFFFFF82,0xFFFFFF83,0xFFFFFF84,0xFFFFFF85,0xFFFFFF86,0xFFFFFF87,0xFFFFFF88,0xFFFFFF89,0xFFFFFF8A,0xFFFFFF8B,0xFFFFFF8C,0xFFFFFF8D,0xFFFFFF8E,0xFFFFFF8F,0xFFFFFF90,0xFFFFFF91,0xFFFFFF92,0xFFFFFF93,0xFFFFFF94,0xFFFFFF95,0xFFFFFF96,0xFFFFFF97,0xFFFFFF98,0xFFFFFF99,0xFFFFFF9A,0xFFFFFF9B,0xFFFFFF9C,0xFFFFFF9D,0xFFFFFF9E,0xFFFFFF9F,0xFFFFFFA0,0xFFFFFFA1,0xFFFFFFA2,0xFFFFFFA3,0xFFFFFFA4,0xFFFFFFA5,0xFFFFFFA6,0xFFFFFFA7,0xFFFFFFA8,0xFFFFFFA9,0xFFFFFFAA,0xFFFFFFAB,0xFFFFFFAC,0xFFFFFFAD,0xFFFFFFAE,0xFFFFFFAF,0xFFFFFFB0,0xFFFFFFB1,0xFFFFFFB2,0xFFFFFFB3,0xFFFFFFB4,0xFFFFFFB5,0xFFFFFFB6,0xFFFFFFB7,0xFFFFFFB8,0xFFFFFFB9,0xFFFFFFBA,0xFFFFFFBB,0xFFFFFFBC,0xFFFFFFBD,0xFFFFFFBE,0xFFFFFFBF,0xFFFFFFC0,0xFFFFFFC1,0xFFFFFFC2,0xFFFFFFC3,0xFFFFFFC4,0xFFFFFFC5,0xFFFFFFC6,0xFFFFFFC7,0xFFFFFFC8,0xFFFFFFC9,0xFFFFFFCA,0xFFFFFFCB,0xFFFFFFCC,0xFFFFFFCD,0xFFFFFFCE,0xFFFFFFCF,0xFFFFFFD0,0xFFFFFFD1,0xFFFFFFD2,0xFFFFFFD3,0xFFFFFFD4,0xFFFFFFD5,0xFFFFFFD6,0xFFFFFFD7,0xFFFFFFD8,0xFFFFFFD9,0xFFFFFFDA,0xFFFFFFDB,0xFFFFFFDC,0xFFFFFFDD,0xFFFFFFDE,0xFFFFFFDF,0xFFFFFFE0,0xFFFFFFE1,0xFFFFFFE2,0xFFFFFFE3,0xFFFFFFE4,0xFFFFFFE5,0xFFFFFFE6,0xFFFFFFE7,0xFFFFFFE8,0xFFFFFFE9,0xFFFFFFEA,0xFFFFFFEB,0xFFFFFFEC,0xFFFFFFED,0xFFFFFFEE,0xFFFFFFEF,0xFFFFFFF0,0xFFFFFFF1,0xFFFFFFF2,0xFFFFFFF3,0xFFFFFFF4,0xFFFFFFF5,0xFFFFFFF6,0xFFFFFFF7,0xFFFFFFF8,0xFFFFFFF9,0xFFFFFFFA,0xFFFFFFFB,0xFFFFFFFC,0xFFFFFFFD,0xFFFFFFFE,0xFFFFFFFF,0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xA,0xB,0xC,0xD,0xE,0xF,0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37};

char* FinalMethod(char actionBits[], char output[]){

    static unsigned int call_count = 0;
    actionBits[call_count] = *output;
    call_count++;
    return output;
}

/*This method is for the do not care stuff , a don't care which is a "-" is represented by "9"*/

void print(char str[], int index, char actionBits[])
{
    /* Get size of str[] Using while loop*/
    int l = 0;
    while (str[l] != '\0')
    {
        ++l;
    }

    if (index == l)
    {

        char* returned_value = FinalMethod(actionBits, str);


        return;
    }

    if (str[index] == '9') {
        // replace '?' by '0' and recurse
        str[index] = '0';
        print(str, index + 1, actionBits);
```

```

        // replace '?' by '1' and recurse
        str[index] = '1';
        print(str, index + 1, actionBits);

        // No need to backtrack as string is passed
        // by value to the function
    }
    else
        print(str, index + 1, actionBits);
}

/*This method is for getting all the possible combinations when there is a don't care/
    skipped levels.
  ---= 000,001,010,011,100,101,110,111
 */
void print_binary(int ints[], int len, int n, int difference, int result, char
    actionBits[]) // this is for sudden jumps
{

    int i = 0;
    int output;
    int number[BDD_OUTPUT_BITS];

    char ss[len];
    if (result == 1) {

        int bit = 1 << difference - 1;

        while (bit) { // if n=4 , 0110

            number[i] = n & bit ? 1 : 0;
            bit >>= 1;
            i++;
        }

        for (int i = 0; i < BDD_OUTPUT_BITS; ++i) {
            ss[i] = number[i] + '0';
        }

        char* returned_value = FinalMethod(actionBits, ss);

        printf("\n");
    }
    else
        return;
}

void printArray(int ints[], int len, char* actionBits)
{

    int k = 0;
    static unsigned int actionCounter = 0;

    char s[len] = { '\0' };
    int n = 0;
    /*If the terminal node is 1*/
    if (ints[len - 1] == 1) {

        for (int i = k; i < BDD_OUTPUT_BITS; ++i) {
            s[i] = ints[i] + '0';
        }

       //cout << "\n CONTROLLER POSSIBLE INPUT " << s << endl;
        print(s, 0, actionBits);
    }
```

```
}

/* function:  printPathsRecur
 *  method for printing all the paths left , it just traverses all the tree till
      terminal nodes
 */
void printPathsRecur(int nodeIndex, int path[], int pathLen, int num, int currentLevel,
     int terminalLevel, int i, char actionBits[])
{
    //node index is the index of the root array
    /* Just an initial condition when we first jump into this METHOD*/
    if (num == 2) {
        /* if after the last user input , I got the output terminal , so all the way
             till terminal would be do not cares*/

        if (i == terminalLevel) {
            /*diff between terminal level and the size input of the user ,because I
                 dont traverse when i dont care*/
            int difference = terminalLevel - BDD_INPUT_BITS;
            /* If i have 4 bits , n=16*/
            int n = 1 << difference, j;

            path[BDD_OUTPUT_BITS + 1] = root[nodeIndex].data;

            for (j = 0; j < n; j++) {
                print_binary(path, BDD_OUTPUT_BITS + 2, j, difference, root[nodeIndex].
                    index, actionBits);
            }
        }
        else {

            printPathsRecur(root[nodeIndex].dashedline, path, pathLen, 0, currentLevel,
                 terminalLevel, i, actionBits);
            printPathsRecur(root[nodeIndex].solidline, path, pathLen, 1, currentLevel,
                terminalLevel, i, actionBits);
        }
    }

    else {

        if (currentLevel == terminalLevel) {

            path[pathLen] = root[nodeIndex].index;

            pathLen++;
            printArray(path, pathLen, actionBits);
        }
        /*  If i did not arrived to terminal*/
        else {
            path[pathLen] = num;
            pathLen++;
            /* if the current level is just after the previous level , continue, which
                means that if this node is in the immediate next level after the
            previous node,no don't cares */
            if (root[nodeIndex].levelnum == currentLevel + 1) { //

                currentLevel++;
                if (currentLevel == terminalLevel) {
                    path[pathLen] = root[nodeIndex].index;

                    pathLen++;
                    printArray(path, pathLen, actionBits);
                    return;
                }
                else
                    printPathsRecur(root[nodeIndex].dashedline, path, pathLen, 0,
```

```
                        currentLevel, terminalLevel, i, actionBits);
                    printPathsRecur(root[nodeIndex].solidline, path, pathLen, 1,
                        currentLevel, terminalLevel, i, actionBits);
                }
                /*If there is a gap,level difference between the currentlevel and the
                    previousLevel ,dont cares*/
                else {

                    /* Difference between levels , if  skipped 2 levels=2 dont cares*/
                    int leveldifference = root[nodeIndex].levelnum - currentLevel - 1;

                    i = root[nodeIndex].levelnum;

                    for (int j = 0; j < leveldifference; j++) {
                        currentLevel++;
                        path[pathLen] = 9;

                        pathLen++;
                        /* I want to have for example 119999900000 so all '9' would be do
                            not cares*/
                    }

                    currentLevel++;
                    /*If I did not reach the terminal level yet*/
                    if (currentLevel < terminalLevel) {

                        printPathsRecur(root[nodeIndex].dashedline, path, pathLen, 0,
                            currentLevel, terminalLevel, i, actionBits);
                        printPathsRecur(root[nodeIndex].solidline, path, pathLen, 1,
                            currentLevel, terminalLevel, i, actionBits);
                    }
                    /*If I reached the terminal level*/
                    else {
                        printPathsRecur(nodeIndex, path, pathLen, 0, currentLevel,
                            terminalLevel, i, actionBits);
                    }
                }
            }
        }
}

/*Function:traverse()
 *Traverses the tree upon the user input , for exmple if stateBits[]=01
 *then this method goes to the dashed line node, then to the solidline node,
 *and when it reaches this node , it simply calls function printPathRecur to
 * traverse the whole tree to Terminal nodes.

 /*
  *nodeIndex=
  *i represents level of the node I am currently in
  *currentLevel
  *previous level is the level I should be in
  terminal level is the level of the terminal
  */

int traverse(int stateBits[], int nodeIndex, int i, int currentLevel, int terminalLevel
    , int leveldifference, char actionBits[])
{

    /*if i finished traversing on nodes upon the user input, so  here i finished going
        on the nodes that was given*/

    if (currentLevel >= BDD_INPUT_BITS) {

        int path[1000];
        printPathsRecur(nodeIndex, path, 0, 2, currentLevel, terminalLevel, i,
```

```
            actionBits);
    }

    /*If the stateBit entered is 0 which represent the dashed line node of the current
        node */
    else if (stateBits[currentLevel] == 0) {

        /* if the current level is just after the previous level , continue, which
            means that if this node is in the immediate next level after the
    previous node,no don't cares */
        if (root[nodeIndex].levelnum == currentLevel + 1) {

            i = root[root[nodeIndex].dashedline].levelnum;
            currentLevel++;
            traverse(stateBits, root[nodeIndex].dashedline, i, currentLevel,
                terminalLevel, leveldifference, actionBits);
        }

        /*If there is a gap,level difference between the currentlevel and the
            previousLevel ,dont cares*/
        else {

            int leveldifference = root[root[nodeIndex].dashedline].levelnum -
                currentLevel - 1; // difference between levels , if  skipped 2 levels=2
                 dont cares
            i = root[root[nodeIndex].dashedline].levelnum; //jump as I don not care
                here , so now i is the level of the node i am in

            currentLevel = currentLevel + leveldifference + 1;
            if (currentLevel >= BDD_INPUT_BITS) {
                currentLevel = BDD_INPUT_BITS;
            }
            traverse(stateBits, root[nodeIndex].dashedline, i, currentLevel,
                terminalLevel, leveldifference, actionBits);
        }
    }

    /*Else If the stateBit entered is 1 which represent solid line node of the current
        node */
    else if (stateBits[currentLevel] == 1) {
        /* if the current level is just after the previous level , continue, which
            means that if this node is in the immediate next level after the
    previous node,no don't cares */
        if (root[nodeIndex].levelnum == currentLevel + 1) {
            i = root[root[nodeIndex].solidline].levelnum;
            currentLevel++;

            traverse(stateBits, root[nodeIndex].solidline, i, currentLevel,
                terminalLevel, leveldifference, actionBits);
        }
        /*If there is a gap,level difference between the currentlevel and the
            previousLevel ,dont cares*/
        else {

            int leveldifference = root[root[nodeIndex].solidline].levelnum -
                currentLevel - 1;

            i = root[root[nodeIndex].solidline].levelnum; //jump as I don not care here
                 , so now i is the level of the node i am in

            currentLevel = currentLevel + leveldifference + 1;
            if (currentLevel >= BDD_INPUT_BITS) {
                currentLevel = BDD_INPUT_BITS; // if the level i must end in does not
                    have a node
            }
            traverse(stateBits, root[nodeIndex].solidline, i, currentLevel,
```

```
                    terminalLevel, leveldifference, actionBits);
            }
        }
}

int getControlAction(int stateBits[], char actionBits[], int terminalLevel)
{

    int path[1000];
    traverse(stateBits, 0, 0, 0, terminalLevel, 0, actionBits); //wrong
}
```