High Level Synthesis using Conditional Decision Diagrams

A project report

submitted by

NISHANTH P. P.

in partial fulfilment of the requirements
for the award of the degrees of

BACHELOR & MASTER OF TECHNOLOGY
in
ELECTRICAL ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

MAY 2015

THESIS CERTIFICATE

This is to certify that the thesis entitled **High Level Synthesis using Conditional Decision Diagrams**, submitted by **Nishanth P. P.**, to the Indian Institute of Technology, Madras, for the award of the degrees of **Bachelor & Master of Technology**, is a bonafide record of the research work carried out by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai
Date:

Dr. Nitin Chandrachoodan
Research Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date:

ACKNOWLEDGMENTS

I am indebtedly grateful to Dr. V. Kamakoti for his guidance throughout the project. His invaluable support and constant motivation have proved indispensable for the completion of the project. I would also like to thank Dr. Nitin Chandrachoodan for providing necessary help and support throughout the project.
I would like to thank Vikas Chauhan for working together with me in bringing this project to fruition. I would also like to thank all the members of RISE lab, especially Neel Gala, Arjun Menon and Arnab Roy, for helping and guiding us throughout the project.
I would like to thank my family and friends for their invaluable support, constant encouragement and unconditional love. I am eternally grateful to them for the same.

ABSTRACT

Decision diagrams are compact graphical representations of Boolean functions mainly used for applications in hardware design, simulation, and formal verification. This report proposes a new type of decision diagram called the Conditional Decision Diagram (CDD), based on the functionally complete, efficient and partially unique Assignment Decision Diagram. We propose an end-to-end solution using CDDs that can generate an optimized gate-level netlist directly from an input hardware description in Chisel, an embedded domain-specific language (DSL) in Scala. Our proposed CDDs retain the unique capabilities of ADDs which are not offered by traditional representations, namely low syntactic variance and estimation of layout quality metrics during synthesis. We propose an algorithm to convert an input ADD netlist to its CDD representation and using a few rules we have developed, generate a minimized CDD representation and convert it back into the ADD netlist. We can hence optimize the number of ADDs in the hardware design.

Contents

List of Figures

GLOSSARY

Chisel	An open-source hardware construction language developed at the University of California at Berkeley that supports advanced hardware design using highly parametrized generators and layered domain-specific hardware languages.
Logic minimization	The problem of obtaining the smallest logic circuit (Boolean formula) that represents a given Boolean function or truth table

ABBREVIATIONS

AST	Abstract Syntax Tree
ADD	Assignment Decision Diagram
BDD	Binary Decision Diagram
CDD	Conditional Decision Diagram
CHISEL	Constructing Hardware in a Scala-Embedded Language
DSL	Domain-Specific Language
FSMD	Finite-State Machine with Datapath
LUT	Look-Up Table
ROCDD	Reduced Ordered Conditional Decision Diagram

Chapter 1

Introduction

Today's VLSI technology enables us to build large, highly complex systems containing billions of transistors on a single chip. To exploit this technology, designers need sophisticated Computer Aided Design tools that enable them to efficiently manage billions of transistors.

In the past few years, logic synthesis has become an integral part of the design process, leading to an evolution in methodology from "capture-and-simulate" to "describe-and-synthesize". The new methodology's advantage is that it allows us to describe a design in a purely behavioral form, devoid of implementation details, and then to synthesize the design structure with CAD tools.

Designers can apply the "describe-and-synthesize" methodology on various levels of abstraction, namely the gate level, FSM level and RTL level with flow charts and data-flow graphs. High-level synthesis is a sequence of tasks that transforms a behavioral representation into an RTL design. The design consists of functional units such as ALUs and multipliers, storage units such as memories and register files, and interconnection units such as multiplexers and buses.

The main advantages of high-level synthesis are productivity gains and better design space exploration. It achieves productivity gains by moving the design process to higher abstraction levels, where designers can specify, model, verify, synthesize, simulate, and debug designs in lesser time. The automation provided by high-level synthesis ensures a more systematic and efficient search of the large design spaces created by the shift to higher abstraction levels.

In this project, we are trying to build an end-to-end solution to generate a final gate-level netlist that would automatically generate optimal, syntactically invariant hardware from a given description in Chisel (a Hardware Construction Language based on Scala being developed by University of California at Berkeley).

We use Assignment Decision Diagrams proposed in [1] to ensure syntactic invariance and convert the resulting netlist into our proposed CDD representation. We have successfully tested converting sample Chisel programs into ADD representation using a compiler phase which modifies the AST of the Scala compiler using a plugin and verified the functional equivalence of the hardware designs with and without using ADDs using FormalPro. We also tried to convert a five-stage RISC-V processor code written in Chisel to its ADD representation, and have fixed numerous bugs in the compiler plugin. In order to minimize the number of ADDs in the design, we run optimizations at the high level of abstraction provided by CDD using our algorithm and then convert the minimized CDD representation back into the ADD netlist. The resulting netlist is guaranteed to have lesser, if not, the same number of ADDs as compared to the input netlist, while providing the same functionality.

The rest of this report is organized as follows: Chapter 2 explains the concept of ADDs and why they were used for high-level synthesis of the hardware design. Chapter 3 introduces the concept of CDDs and the algorithm used to convert the ADD netlist to a CDD representation, minimize the CDD and convert it back to the ADD netlist.

Chapter 2

Assignment Decision Diagrams

This chapter summarizes the concept of Assignment Decision Diagrams proposed by Gajski et al. in [1] and explains its advantages pertaining to high-level synthesis. This chapter contains some content verbatim from the above mentioned paper.

2.1 Introduction

The ADD representation was developed in order to encapsulate the functionality of a described hardware in a simple, precise and unique manner. These three objectives are treated with utmost importance because of the following reasons:

- The representation's uniqueness allows synthesis tools to be independent of syntactic variances usually present in the input description. The ADD should be able to depict the most parallel representation of the input description to satisfy the uniqueness property.
- The representation should also consist of parts that reflect the description's semantics instead of syntactic constructs. Each and every part of the representation should have no direct relationships with language constructs. This is referred to as the preciseness of the representation.

- A representation is simple if it consists of a few number of different object types and relationships between each object type. Such representations can simplify synthesis algorithms because the algorithms have to manage small number of objects. Since most of the synthesis algorithms are topology graph based, the representation for a synthesis system has to be a form of topology graph. A simple representation is thus, a graph that consists of a small number of different types of nodes and edges.

Digital systems can be classified into sequential and and combinatorial systems. Functionality of the sequential and and the combinatorial systems can be described by the Finite-State Machine and Datapath model (FSMD) as shown in Figure 2.1(a).

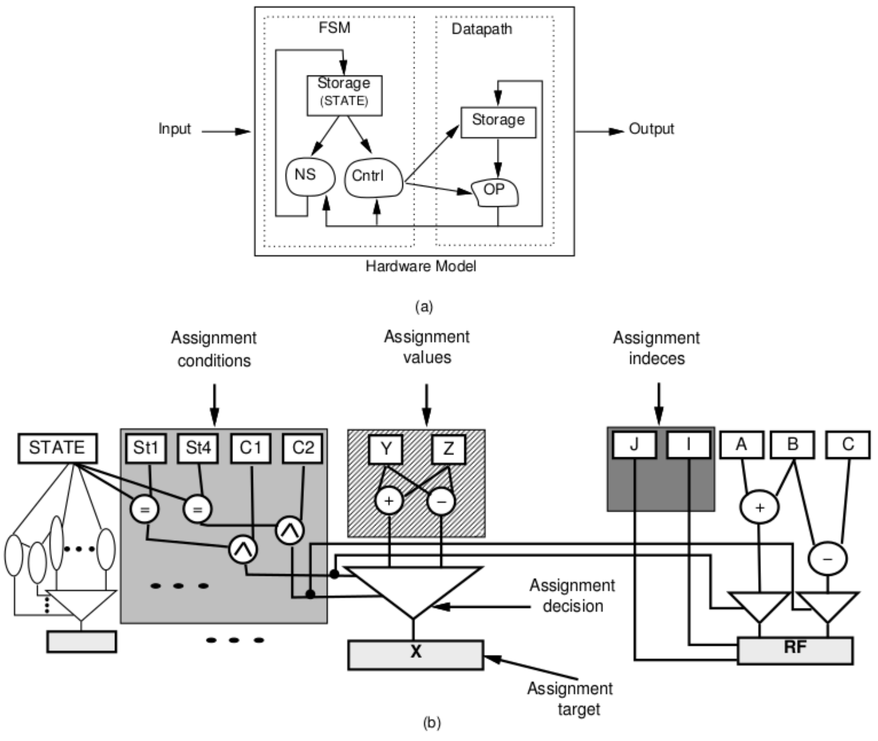


Figure 2.1: The Assignment Decision Diagram: (a) FSMD model, (b) the ADD [1]

Note that in the case of a combinatorial system, storage units will not be present in the model. The FSMD model can be viewed as assignments of values to storage units (either the storage units in the datapath or the state variable) and output ports based on the state of the system along with certain conditions. These conditional assignments are represented in the Assignment Decision Diagram shown in Figure 2.1(b). The state variable is represented in the ADD in the same way as any other variable.

2.2 High-level synthesis using ADDs

The first task in high-level synthesis is to compile the input description into an internal representation that is usually in a form of a topological graph. The compilation is usually accomplished by a one-to-one mapping of the input description into the internal representation, i.e., each language construct in the description is realized with a particular topology of nodes in the representation, and hence different descriptions could lead to disparate representations, even if they are semantically equivalent. Compilers generate graphs with different topologies for different descriptions, and even though the graphs themselves may be semantically the same, synthesis algorithms, which are topology based, would produce different hardware for different topologies, as illustrated in Figure 2.2.

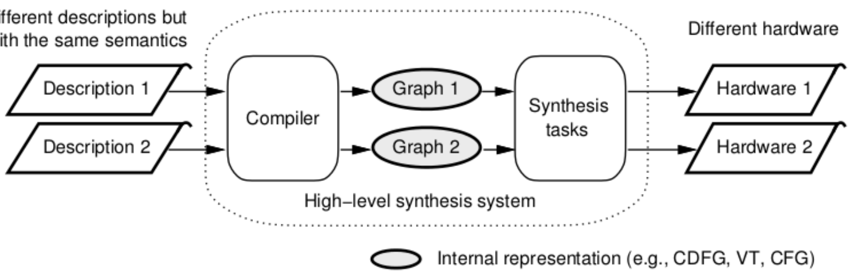

Different descriptions but with the same semantics
Description 1
Description 2
Compiler
Graph 1
Graph 2
Synthesis tasks
High–level synthesis system
Different hardware
Hardware 1
Hardware 2
Internal representation (e.g., CDFG, VT, CFG)

Figure 2.2: A general high-level synthesis approach [1].

ADDs represent different descriptions having the same semantics in a “unique” topology. They use the most parallel, and not the most sequential representation of the input description to be the “unique” representation because it does not contain implicit sequentiality found in the description. Thus, ADDs can depict the most parallel representation of any given input description. Traditional representations can not provide such capability because their constructs inherit the sequentiality from the description.

After defining ADDs, we develop a compilation scheme from the input description into the new representation. The compilation transforms/converts a given description into its most parallel representation and, at the same time, resolves the discrepancies that are caused by the ordering and grouping of conditional branches and/or computations. As a result, different descriptions that contain such discrepancies can be transformed into a “unique” graph so that the result obtained from synthesis tasks is consistent. The ADD approach is illustrated in Figure 2.3.

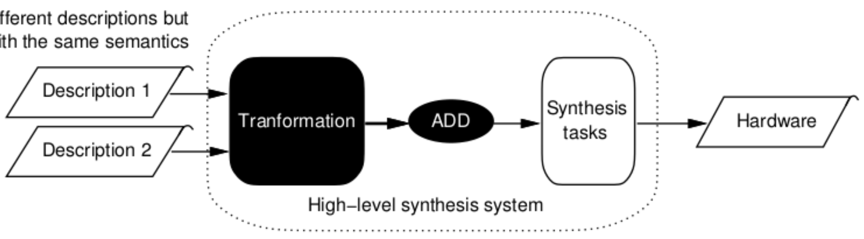


Figure 2.3: High-level synthesis using Assignment Decision Diagrams [1].

Our ultimate objective is to design all combination parts of a hardware description using only ADDs and completely devoid of gates, since the ADD is functionally complete and can replace any gate in the hardware design.

2.3 Representing ADDs in Chisel

In this section, we explain the structure of an Assignment Decision Diagram and the equation that governs it, along with a few well known Chisel constructs represented as ADDs, namely when-elsewhen-otherwise, switch, mux, muxcase and muxlookup.

2.3.1 ADD representation

The Assignment Decision Diagram is a MUX-like structure with conditions having a priority order. A 2-input ADD has two inputs i_1, i_2, two conditions C_1, C_2, with C_1 having higher priority over C_2, and a default value, D, as shown in Figure 2.4.

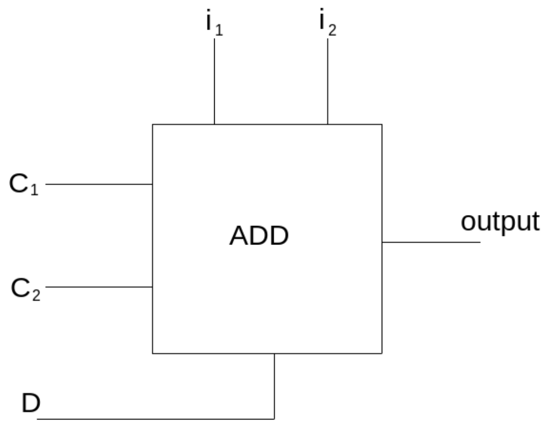


Figure 2.4: A 2-input Assignment Decision Diagram

The output is assigned one of the inputs or the default value depending on the values of the conditions, given by

$$Output = C_1 i_1 + \overline{C}_1 C_2 i_2 + \overline{C}_1 \overline{C}_2 D \tag{2.1}$$

2.3.2 The when-elsewhen-otherwise construct

The when-elsewhen-otherwise construct is used in the behavioral description to produce sequential condition-branching effect. If the condition in the when part is true, the operations accompanying it are executed, if not, then the condition in the elsewhen part is checked. If it is true, the operations associated with that elsewhen condition are executed, and if not, the next elsewhen condition is checked and so on. Optionally, there can exist an otherwise condition whose operations are executed only if all the when and elsewhen conditions don't hold true.

```
/* Chisel excerpt of when-elsewhen-otherwise */
```

```
when (io.opcode === UInt(0)) {
io.output := io.a + io.b    // ADD
} .elsewhen (io.opcode === UInt(1)) {
io.output := io.a - io.b    // SUB
} .otherwise {
io.output := io.a           // PASS A
}
```

The ADD representation of the above description is given in Figure 2.5. Here, the first condition is $C_1 \equiv (io.opcode === UInt(0))$ and the second condition is $C_2 \equiv (io.opcode === UInt(1))$. The inputs are $i_1 \equiv io.a+io.b$ and $i_2 \equiv io.a-io.b$ respectively and the default value, $D \equiv io.a$. The output is given by *io.output*.

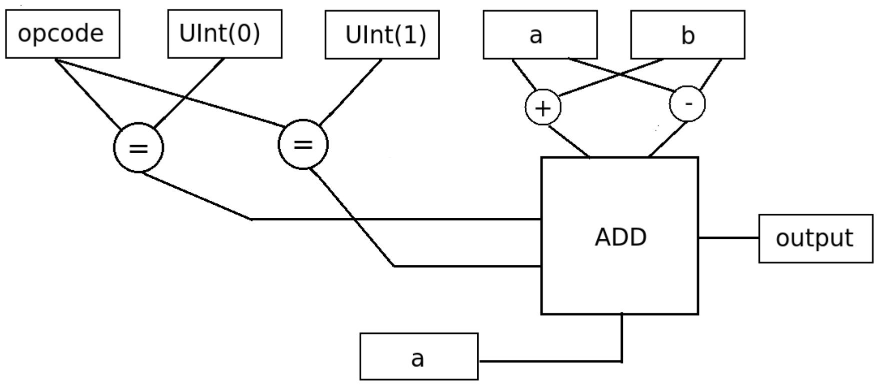


Figure 2.5: ADD realization of Chisel when-elsewhen-otherwise and switch-is constructs

2.3.3 The switch-is construct

The switch construct is similar to the when-elsewhen-otherwise construct and provides multi-way branching capability to the sequential description. Operations in a branch are executed if the accompanying condition evaluates to true.

```
/* Chisel excerpt of switch-is */
```

```
io.output := io.a               // PASS A
switch (io.opcode) {
   is (UInt(0)) {
    io.output := io.a + io.b    // ADD
 } is (UInt(1)) {
    io.output := io.a - io.b    // SUB
 }
}
```

The ADD representation of the above description is the same as that of when, i.e. Figure 2.5. The hardware is hence bound to be the same, since the excerpts are performing the same function.

2.3.4 The Mux construct

Chisel has the Mux function in-built. It takes in as input a Boolean *select*, a value to be assigned to the output when *select* is **True** and a value to be assigned to the output when *select* is **False**.

```
/* Chisel excerpt of Mux */
io.output := Mux((io.opcode === UInt(0)), io.a,
             Mux((io.opcode === UInt(1)), io.b,
                                          io.c))    // Default
```

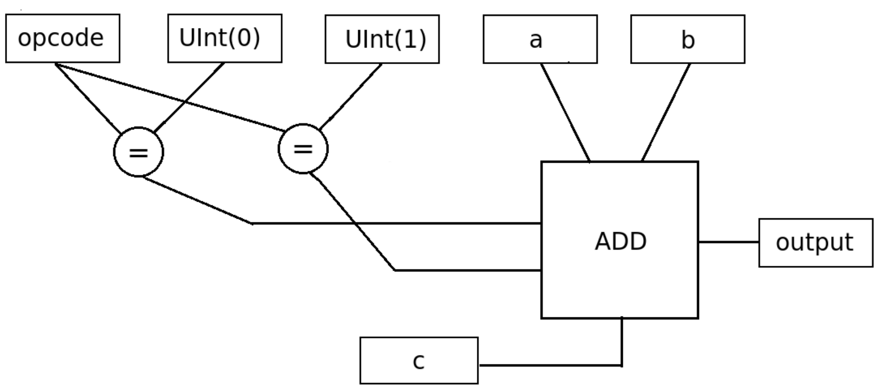


Figure 2.6: ADD realization of Chisel Mux, MuxCase and MuxLookUp constructs

The ADD representation of the above description is shown in Figure 2.6. Here, the first condition is $C_1 \equiv (io.opcode === UInt(0))$ and $C_2 \equiv (io.opcode === UInt(1))$. The inputs are $i_1 \equiv io.a$ and $i_2 \equiv io.b$ respectively and the default value, $D \equiv io.c$. The output is given by $io.output$.

2.3.5 The MuxCase construct

Chisel provides MuxCase which is an n-way Mux where each condition/value is represented as a tuple in a Scala array and where MuxCase can be translated into a Mux expression as shown below:

`MuxCase(default, Array(c1 -> a, c2 -> b, ...))` is equivalent to
`Mux(c1, a, Mux(c2, b, Mux(..., default)))`

```
/* Chisel excerpt of MuxCase */

io.output := MuxCase(io.c, Array(                    // Default
             (io.opcode === UInt(0)) -> io.a,
             (io.opcode === UInt(1)) -> io.b
             ))
```

The ADD representation of the above description is the same as that of Mux shown in Figure 2.6.

2.3.6 The MuxLookUp construct

Chisel also provides MuxLookup which is an n-way indexed multiplexer. It can also be translated into a Mux expression as shown below:

`MuxLookup(idx, default, Array(v1 -> a, v2 -> b, ...))` is equivalent to

```
Mux((idx===v1), a, Mux((idx===v2), b, Mux(..., default)))
```

```
/* Chisel excerpt of MuxLookUp */

io.output := MuxLookUp(io.opcode, io.c, Array(                  // Default
                                UInt(0) -> io.a,
                                UInt(1) -> io.b))
```

The ADD representation of the above description is again the same as that of Mux shown in Figure 2.6.

Chapter 3

Conditional Decision Diagrams

This chapter proposes a new type of data structure based on the Assignment Decision Diagram called the Conditional Decision Diagram, in order to optimize the number of ADDs at the same abstraction level as the ADD in a given hardware design.

3.1 Introduction

The Conditional Decision Diagram is based on the Assignment Decision Diagram and hence has all of its fundamental properties. It is modeled as a Directed Acyclic Graph with edges connecting the output to the input conditions and values. Figure 3.1(a) shows the ADD and Figure 3.1(b) shows the corresponding CDD. For simplicity, we ignore the default value of the ADD and use our algorithm, the CDD logic minimizer, only on those CDDs which have the same value for default. Also, since we are considering only those CDDs which have the same default value, Equation 2.1 reduces to

$$Output = C_1V_1 + \overline{C}_1C_2V_2 \tag{3.1}$$

where V_1, V_2 are the values to be assigned to the output and C_1 and C_2 are the conditions, with C_1 having higher priority over C_2.

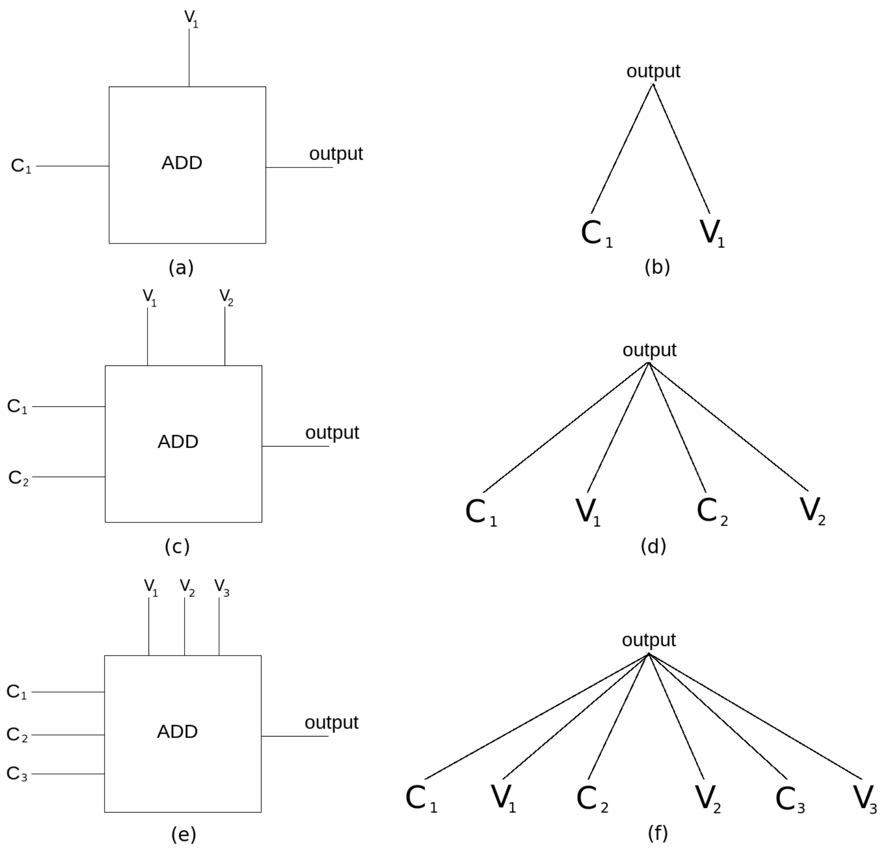


Figure 3.1: (a) & (b) A single input ADD and its corresponding CDD; (c) & (d) A 2-input ADD and its corresponding CDD; (e) & (f) A 3-input ADD and its corresponding CDD

Our objective is to integrate the CDD logic minimizer into the Chisel compiler such that it takes in as input the ADD netlist generated by our plugin and sends as output the minimized ADD netlist back to the plugin, which then sends the optimized netlist in the required format to the Scala compiler to generate Verilog code corresponding to the input description in Chisel. We use five rules to achieve the logic minimization, while preserving the functionality of the CDDs while applying each rule. The functionality of the modified netlist is thus guaranteed to be the same as that of the input netlist. The rules are explained in detail in the following section. Our high-level synthesis approach is illustrated in Figure 3.2 below.

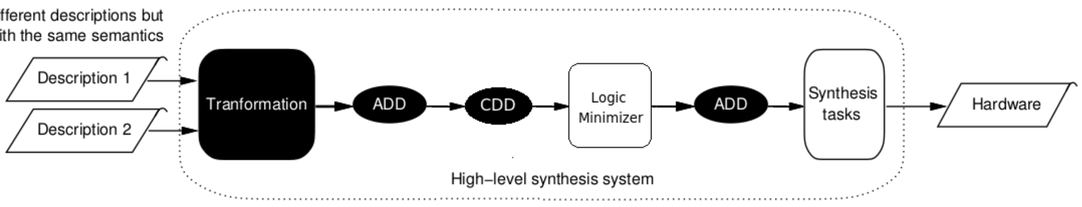


Figure 3.2: Our high-level synthesis approach.

The Conditional Decision Diagrams heavily depend on the variable ordering chosen and the degrees of minimization can vary greatly depending on it. But for a fixed ordering, CDDs are canonical, i.e. for a certain expression with a certain chosen variable ordering, we can only have one CDD representing it.

3.2 Logic Minimization

We need to first look at where minimizations can be performed over an ADD netlist and then we will show our approach to perform the minimization.

3.2.1 Scope for minimization

Consider an excerpt of code from a Chisel program implemented using ADDs.

```
/* Chisel excerpt for logic minimization */

io.output := io.e                  // Default
when (io.opcodea === UInt(0)) {
    when (io.opcodeb === UInt(1)) {
        when (io.opcodec != UInt(2)) {
            when (io.opcoded === UInt(3)) {
                io.output := io.b - io.c
           } .elsewhen (io.opcoded != UInt(3)) {
                io.output := io.a + io.b
            }
        }
    } .elsewhen (io.opcodeb != UInt(1)) {
        when (io.opcodec === UInt(2)) {
            when (io.opcoded != UInt(3)) {
                io.output := io.a + io.b
            }
        }
    }
} .elsewhen (io.opcodea != UInt(0)) {
    when (io.opcodeb === UInt(1)) {
        when (io.opcodec === UInt(2)) {
            io.output := io.a - io.c
        }
    } .elsewhen (io.opcodeb != UInt(1)) {
        when (io.opcodec != UInt(2)) {
            when (io.opcoded === UInt(3)) {
                io.output := io.a + io.b
            }
        }
    }
}
io.output := io.e                   // Default
when (io.opcodea === UInt(0)) {
    when (io.opcodeb === UInt(1)) {
        when (io.opcodec === UInt(2)) {
            when (io.opcoded === UInt(3)) {
                io.output := io.a - io.b
            } .elsewhen (io.opcoded != UInt(3)) {
                io.output := io.a + io.b
            }
        }
    }
} .elsewhen (io.opcodea != UInt(0)) {
    when (io.opcodeb === UInt(1)) {
        when (io.opcodec != UInt(2)) {
            io.output := io.a + io.b
        }
    }
}
```

The same output is being assigned different values depending on the conditions and it has the same default value in both instances shown above. The ADD representation of the above excerpt would be as shown in Figure 3.3.

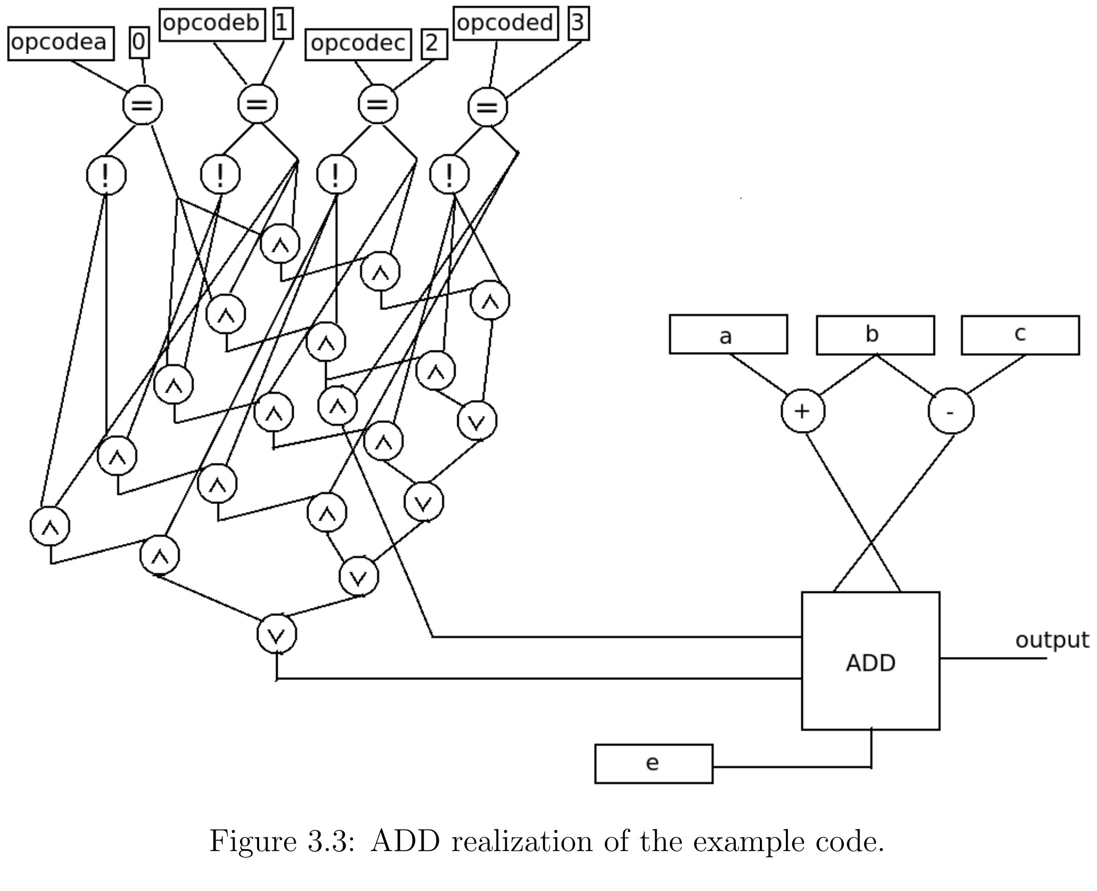


Figure 3.3: ADD realization of the example code.

Note that we are realizing the combinatorial part on the left using just ADDs and not gates. This is where we can use logic minimization. A few basic gates and their CDD representations are given in Figure 3.4 below.

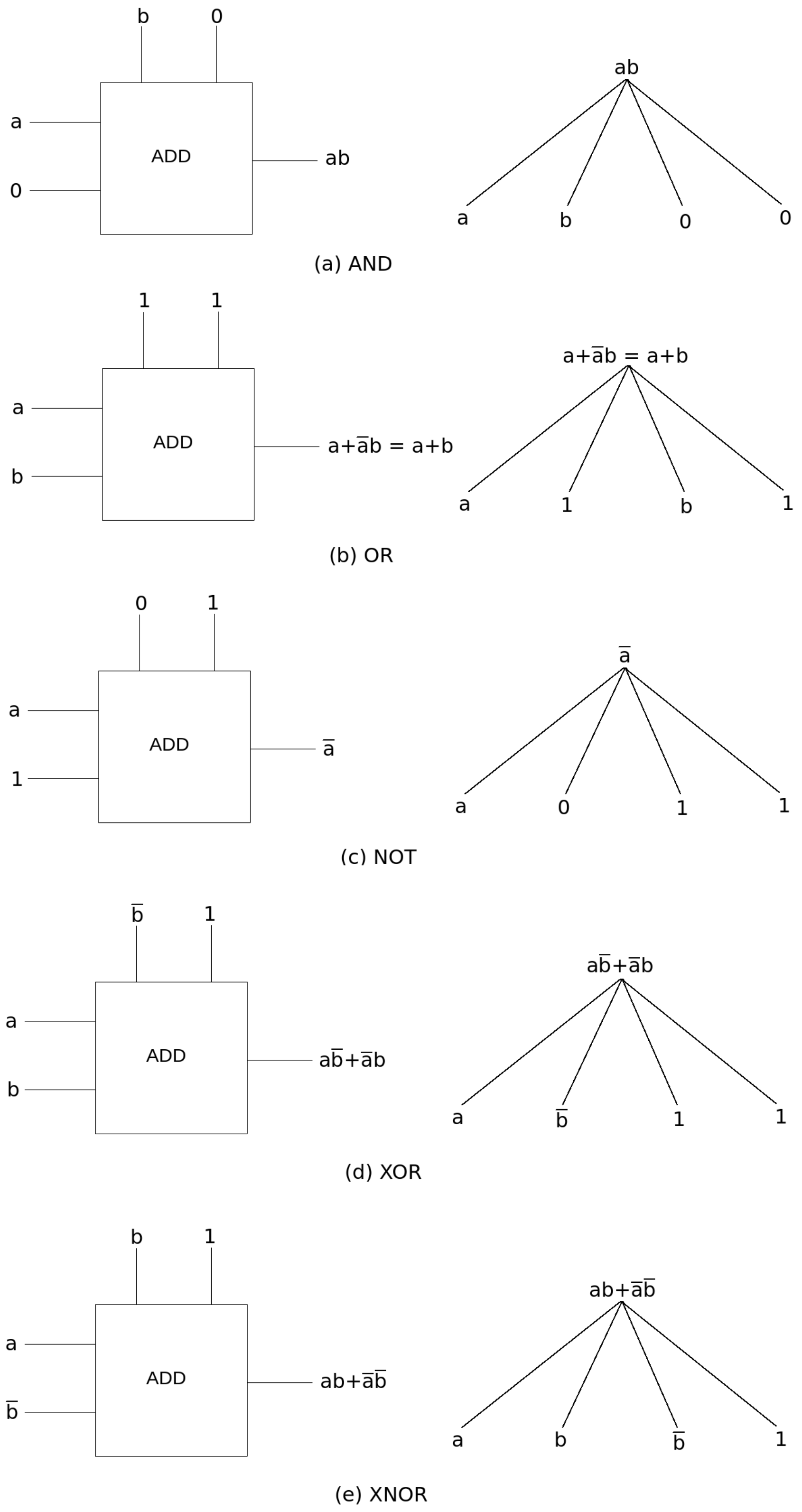


Figure 3.4: Gates realized using ADDs along with their CDD representation.

Using these CDD representations, we can prepare the ADD netlist for the code excerpt realized in Figure 3.3. The netlist is shown below in Figure 3.5.
Let $a \equiv (io.opcodea === UInt(0))$, $b \equiv (io.opcodeb === UInt(1))$, $c \equiv (io.opcodec === UInt(2))$ and $d \equiv (io.opcoded === UInt(3))$. Now we have to minimize the Boolean equation

$$F = ab\bar{c}\bar{d} + a\bar{b}c\bar{d} + \bar{a}\bar{b}\bar{c}d + \bar{a}b\bar{c} + abc\bar{d}$$

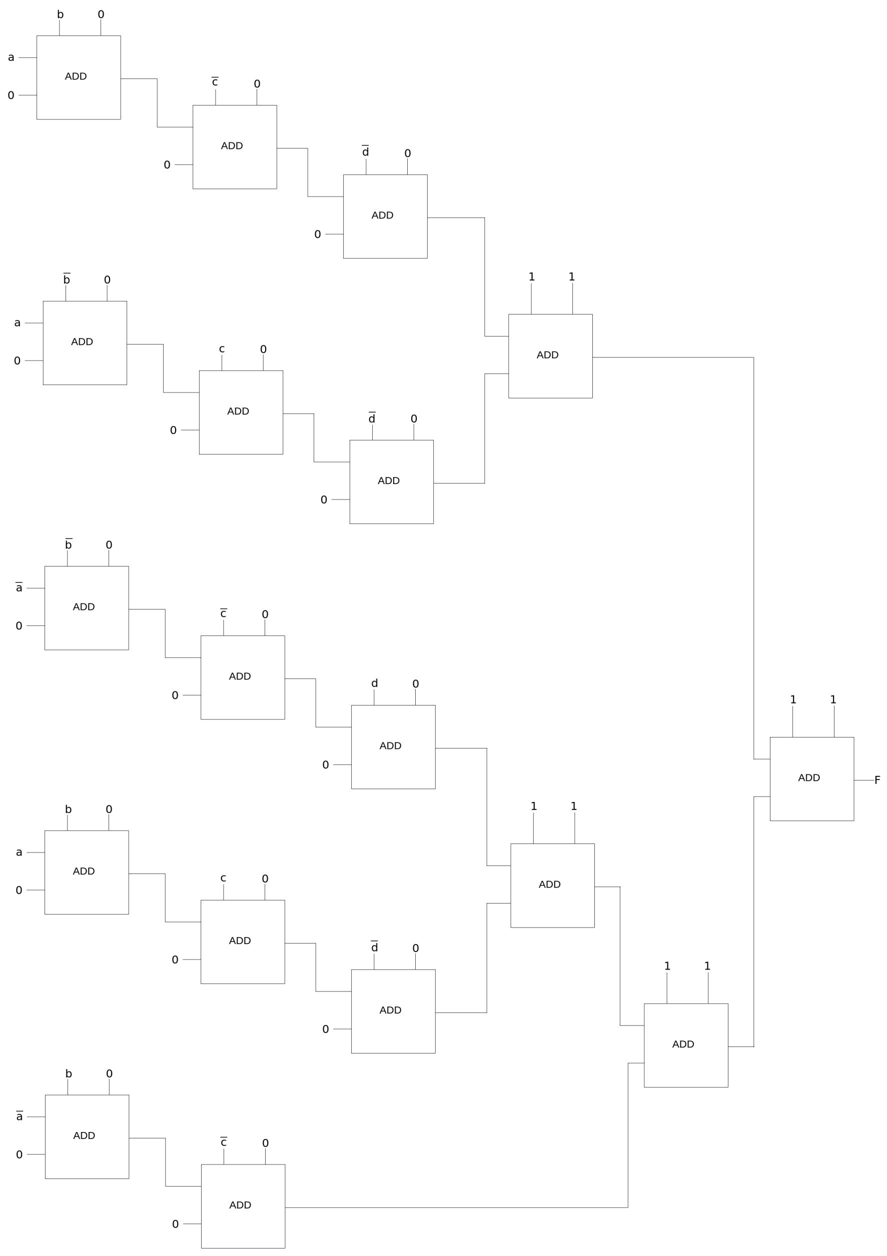


Figure 3.5: The ADD netlist of the combinatorial part of Figure 3.3.

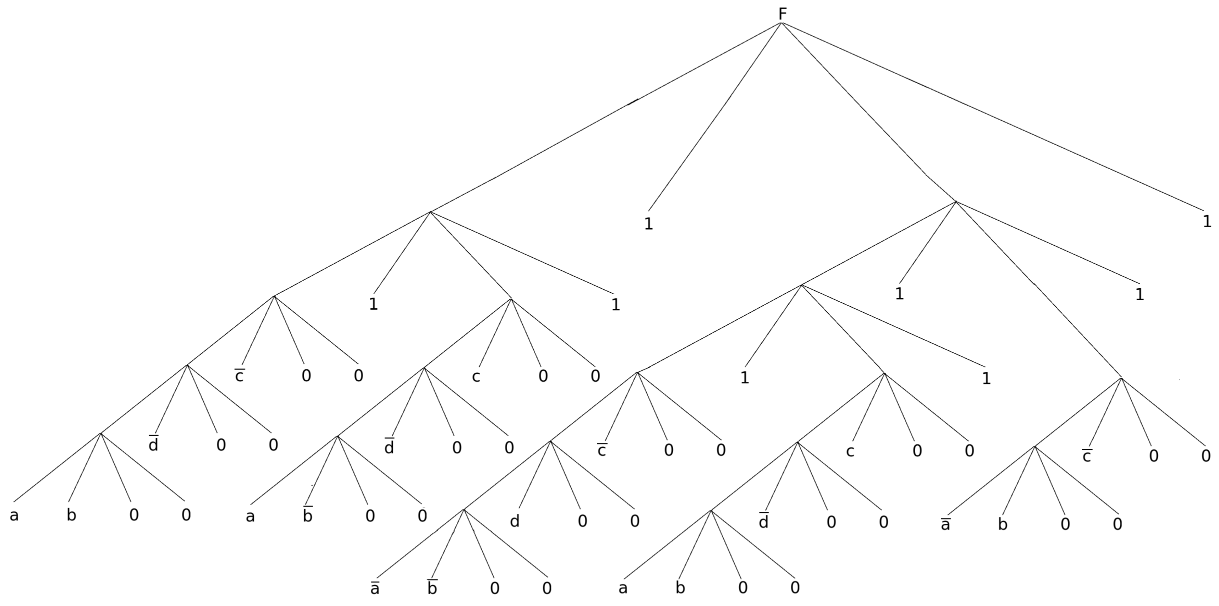


Figure 3.6: The CDD representation of the ADD netlist in Figure 3.5.

3.3 Minimization Rules

We have developed eight rules using which we perform the logic minimization. They can be applied on CDDs assigning the same value to the output while having the same value for default. We define an ROCDD, Reduced Ordered Conditional Decision Diagram, as a decision diagram in which a particular ordering of literals has been chosen for the minterms of the boolean expression, along with the following rules being applied over it in order to simplify the Conditional Decision Diagram. From now onwards, we shall use the term CDD to refer to ROCDD.

3.3.1 Rule 1

If a CDD is assigning the same value to the output and the conditions are complements of each other, then the CDD can be removed and replaced by that value alone, without any loss in functionality.

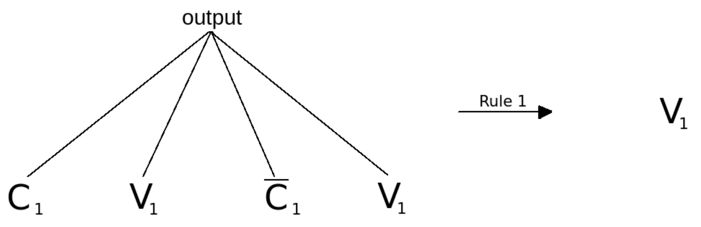


Figure 3.7: Rule 1

3.3.2 Rule 2

If we have a CDD where both the conditions and values are repeated more than once, we replace the CDD with just one instance of the condition-value pair.

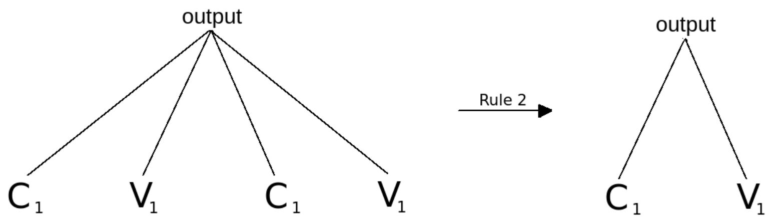


Figure 3.8: Rule 2

3.3.3 Rule 3

If an CDD is performing an AND operation over two OR CDDs which have the same conditions but one of them is complemented in one of the OR CDDs, the three CDDs can be replaced by an ADD having just the condition that was not complemented.

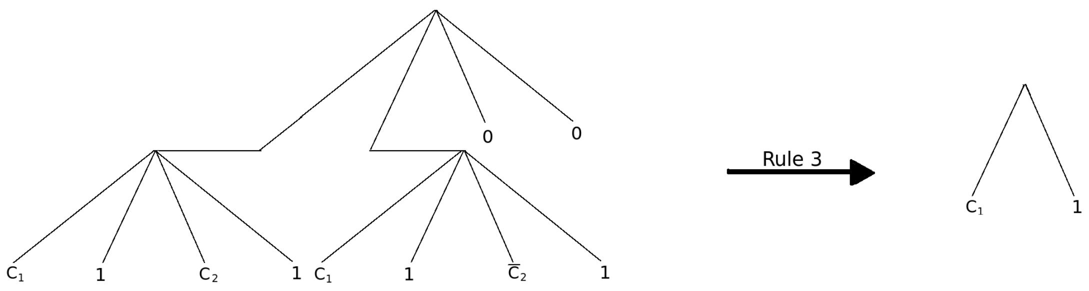


Figure 3.9: Rule 3

3.3.4 Rule 4

If a CDD is performing an OR operation over two AND CDDs which have the same conditions but one of them is complemented in one of the AND CDDs, the three CDDs can be replaced by an ADD having just the condition that was not complemented.

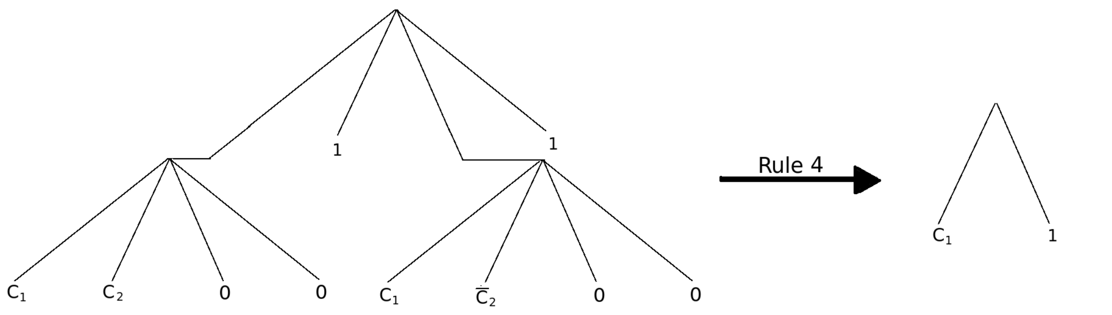


Figure 3.10: Rule 4

3.3.5 Rule 5

If a CDD is performing an OR operation over another CDD which is also performing OR (nested OR), the input size of the parent CDD can be increased by one less than the number of children of the child CDD, in order to accommodate the children of the sibling CDD, and the child CDD can be removed. This helps remove nested ORs so there is no unnecessary hierarchy involved during simplification. After simplifying, all the OR gates having input size more than two can be replaced by nested ORs before converting the CDDs back to the ADD netlist.

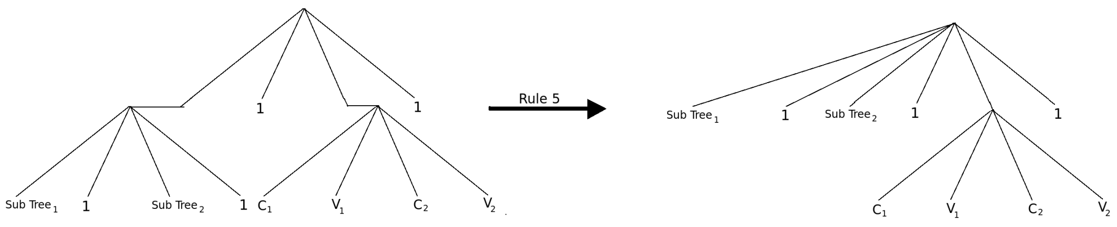


Figure 3.11: Rule 5

3.3.6 Rule 6

We can remove all CDDs succeeding a condition if it is a tautology. Since there exists a priority order for the conditions, we cannot remove the preceding ones.

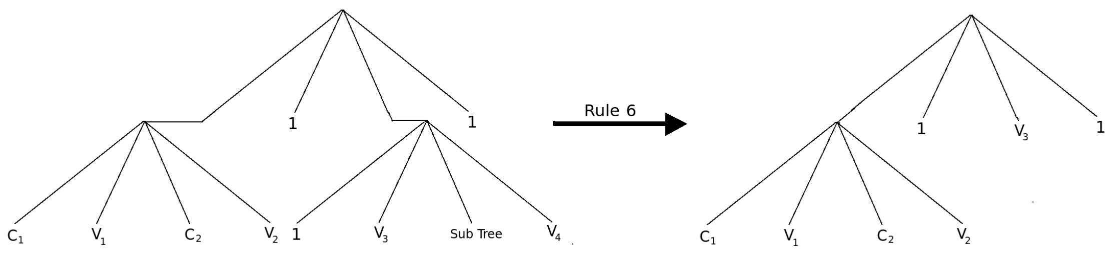


Figure 3.12: Rule 6

3.3.7 Rule 7

The boolean redundancy rule is used to simplify CDDs having conditions as shown in Figure 3.13.

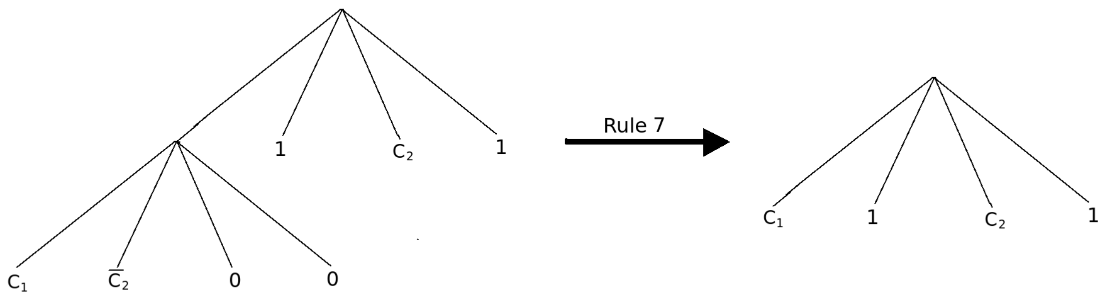


Figure 3.13: Rule 7

3.3.8 Rule 8

We use this rule to branch out at each node, in order to simplify the process of finding minterms which are one literal away from each other.

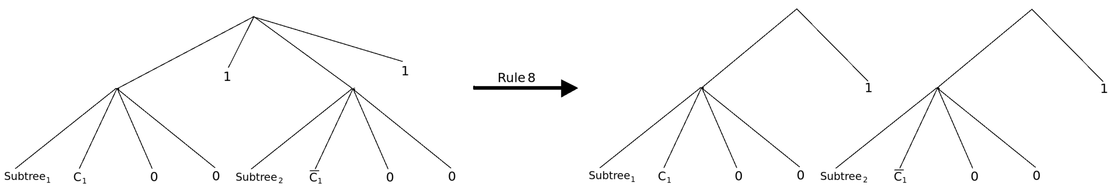


Figure 3.14: Rule 8

3.4 Minimization Algorithm

The ADD netlist, N, that we are provided contains, in each line L, the combinations of conditions, C assigning an output variable V a particular value for that combination of conditions, v and the default value, D, also for that combination of conditions. The algorithm is given in Algorithm 1.

3.4.1 Working of the algorithm

The first part of the algorithm is to identify the combinations of conditions for which the same output variable is being assigned the same value, along with the same value being assigned as default for that combination of conditions. Once we have all the combinations of conditions, we store them in groups H which are identified uniquely by the output variable, the value it is assigned and the default value. Each group H contains minterms of conditions in SOP form. For each H, we can minimize the number of ADDs used to implement it. We first choose a group H_i, add missing literals using a procedure called Reduce so that we have all the literals in each minterm, choose a certain ordering of variables and construct trees for each minterm. We then construct an OR-Tree which has all the minterm-trees together. This OR-Tree is unique for a particular H. We then apply our CDD rules over the OR-Tree in a procedure similar to technology mapping. The CDD obtained after applying all the rules will be having lesser, if not, the same number of minterms as the input ADD, depending on the variable ordering.

To demonstrate with an example, let us consider Figure 3.6. Once we have this as the input, we apply Rule 5 to obtain the CDD shown in Figure 3.15.

```
$N \equiv (L, C, V, v, D)$
Algorithm MinCDD($N$)
    for each line $L \in N$ do
        for each $V_i \in V$ do
            for each $v_j \in v_{V_i}$ do
                for each $D_k \in D_{v_j}$ do
                    $H_m \leftarrow \mathrm{H}_m \cup L$
                end
            end
        end
    end
    for each $H_m \in H$ do
        $BuildCDD(H_m)$
    end
    Procedure BuildCDD($H$)
        for each $C_i \in \mathbb{C}_H$ do
            $\mathbb{C}_H \leftarrow \mathbb{C}_H - C_i$
            $\mathbb{C}_H \leftarrow \mathbb{C}_H \cup Reduce(C_i, Literals(\mathbb{C}_H))$
        end
        while $nterms(minimizedCDD(H)) > nterms(inputCDD)$ do
            $O = RandomOrdering(Literals(\mathbb{C}_H))$
            for each $C_j \in \mathbb{C}_H$ do
                $SkewedTree(C_j, O) \leftarrow BuildSkewedTree(C_j, O)$
                $ORTree(\mathbb{C}_H) \leftarrow ORTree(\mathbb{C}_H) \cup SkewedTree(C_j, O))$
            end
            $Map(Root_{ORTree(\mathbb{C}_H)})$
        end
        Return $minimizedCDD(H)$
    Procedure Map(node $n$)
        if $n \neq leafnode$ then
            while no more rules can be applied do
                Technology Mapping using Rules starting from node
            end
            Map(Children of $n$)
        end
```
Algorithm 1: CDD minimization algorithm

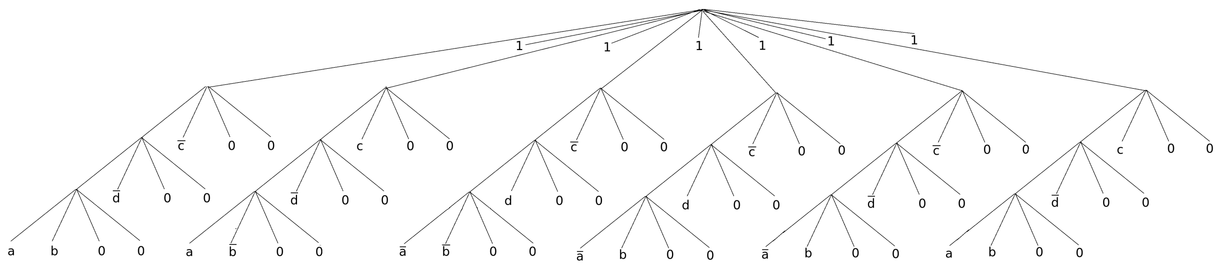


Figure 3.15: CDD obtained after applying Rule 5 over the CDD in Figure 3.6.

After this, we apply Rule 8 over the root node and subsequent nodes until we reach the leaf nodes. The subsequent figures show the branching at each node.

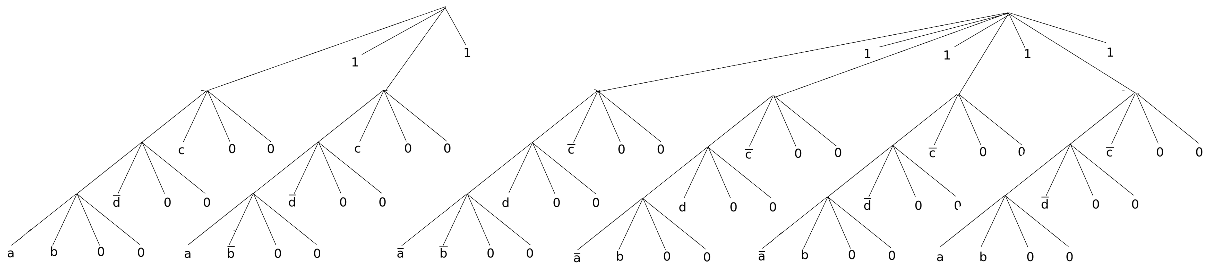


Figure 3.16: Branching at the child of root, node c, by applying Rule 8 over the CDD in Figure 3.15.

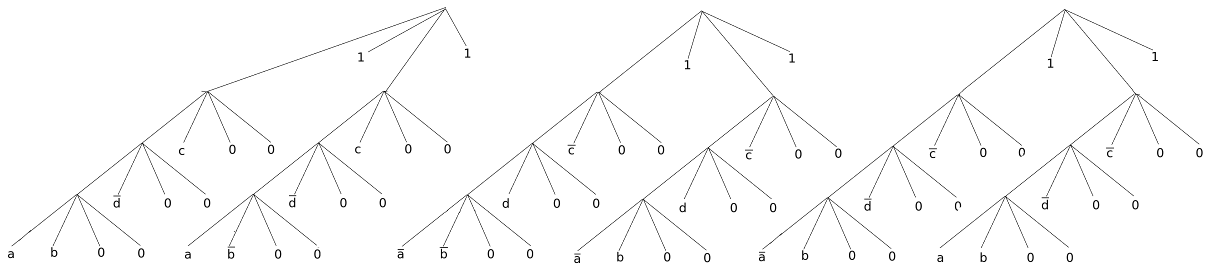


Figure 3.17: Branching at the child of node c, node b, by applying Rule 8 over the CDD in Figure 3.16.

Now since the we reach the leaf nodes, we stop branching and apply Rule 4 on each of the branched trees, resulting in Figure 3.18.

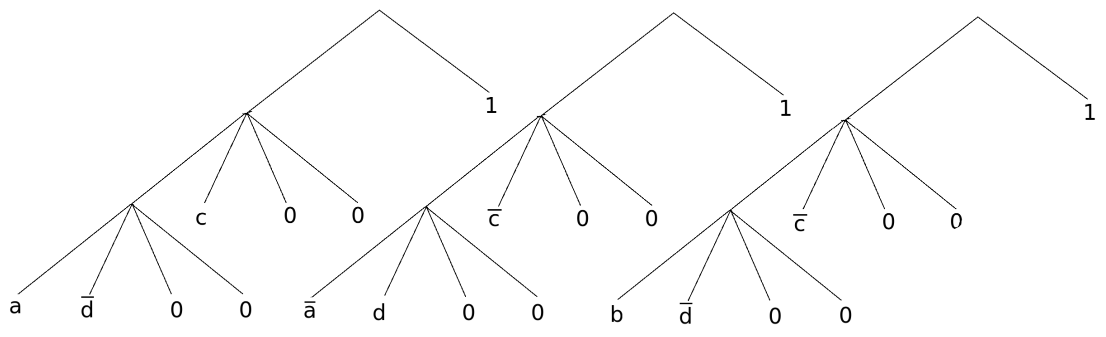


Figure 3.18: Resulting CDD after applying Rule 4 over the CDD in Figure 3.17.

Since no more rules can be applied, we combine the minterm trees using the inverse of Rule 8 i.e. performing OR all the minterm trees and generating back the ADD netlist as shown in Figure 3.19.

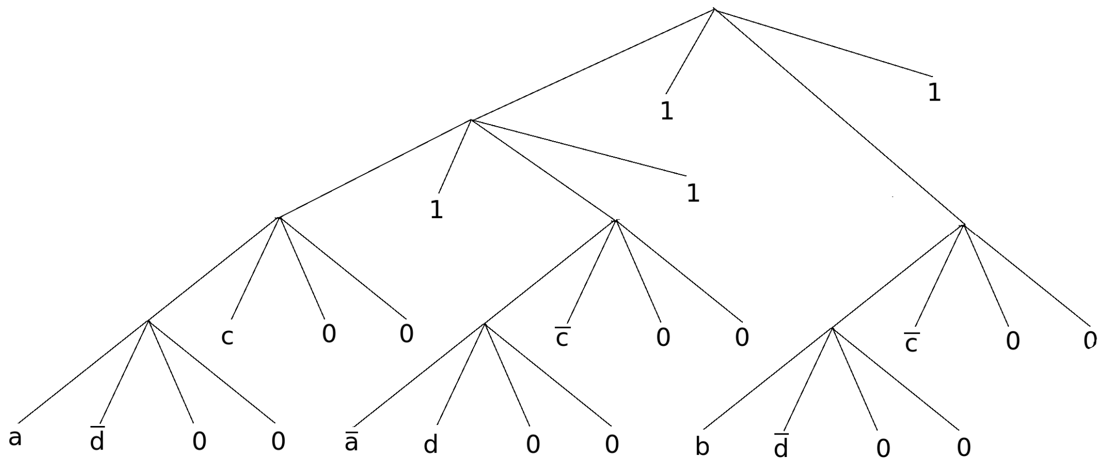


Figure 3.19: Output CDD which will be converted to the ADD netlist.

The resulting CDD obtained can be converted to the ADD netlist by a simple parsing of the nodes in the tree. In the example shown here, the result is optimal, as can be seen from a Karnaugh Map. The simplified ADD netlist is shown in Figure 3.20 below.

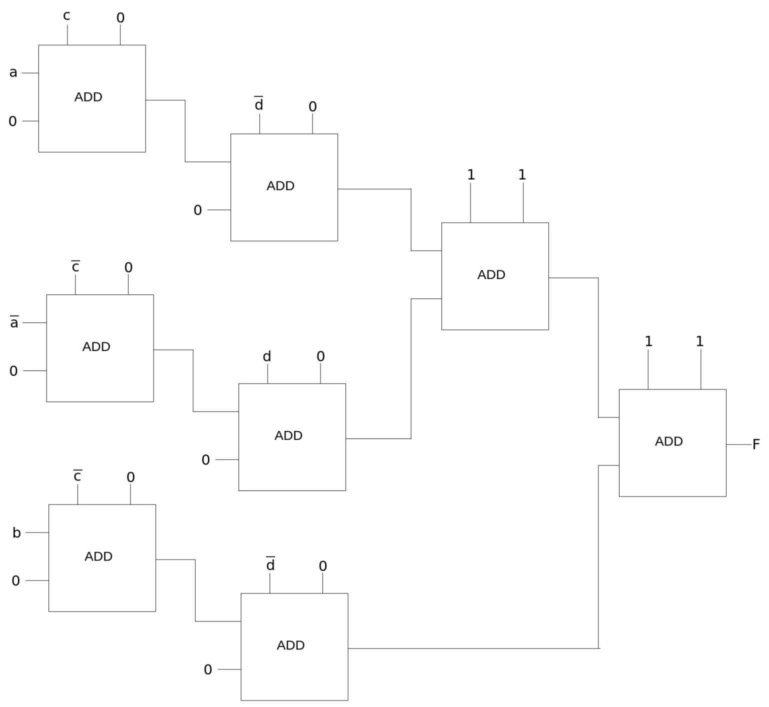
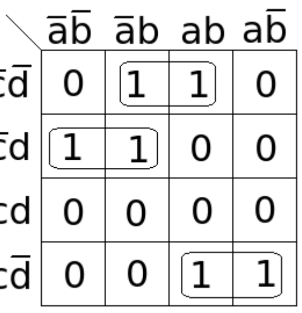


Figure 3.20: Simplified ADD netlist. ADD count: 8.

It only contains 8 ADDs as compared to 18 in the input. The simplified boolean expression is now

$$F = ac\bar{d} + \bar{a}\bar{c}d + b\bar{c}\bar{d}$$

which is the optimal solution as can be seen from the Karnaugh map in Figure 3.21 below.



	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$
$\bar{c}\bar{d}$	0	1	1	0
$\bar{c}d$	1	1	0	0
cd	0	0	0	0
$c\bar{d}$	0	0	1	1

Figure 3.21: Karnaugh map of F.

Chapter 4

Conclusions and Future Work

This report has proposed an new type of decision diagram called the Conditional Decision Diagram to minimize the number of Assignment Decision Diagrams used in a hardware design. We have described how to incorporate it in the ADD netlist and have also shown how the minimization can be performed. This was introduced so as to minimize the number of ADDs used in the hardware design, at the same level of abstraction as the ADD. We have shown that even though the result may not always be optimal and depends on the variable ordering we choose for the CDD, the result obtained for any Boolean function we are provided as input from the ADD representation is guaranteed to be minimized and in the worst case, the same as that of the input.

The properties of this data structure have to be explored in depth and its performance needs to be compared with conventional data structures like BDDs (Binary Decision Diagrams) proposed in [3] and [5]. The algorithm needs to be implemented in the Scala compiler plugin to automate the minimization process and its performance on a substantial number of examples, including the RISC-V cores needs to be benchmarked against BDDs.

REFERENCES

[1] Viraphol Chaiyakul and Daniel D. Gajski. "Assignment Decision Diagram for High-Level Synthesis". In *Technical Report, Information and Computer Science, University of California, Irvine*, pages 92–103, Dec. 12, 1992.

[2] Viraphol Chaiyakul, Daniel D. Gajski, and Loganath Ramachandran. "High-level Transformations for Minimizing Syntactic Variances". In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 413–418, New York, NY, USA, 1993. ACM.

[3] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[4] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. "Chisel: Constructing hardware in a Scala embedded language". In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[5] S.B. Akers. "Binary Decision Diagrams". *Computers, IEEE Transactions on*, C-27(6):509–516, June 1978.

[6] Indradeep Ghosh and Masahiro Fujita. "Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams". In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 43–48, New York, NY, USA, 2000. ACM.

[7] Liang Zhang, M. Hsiao, and I. Ghosh. "Automatic design validation framework

for HDL descriptions via RTL ATPG". In *12th Asian Test Symposium, (ATS) 2003*, pages 148–153, Nov 2003.

[8] D. Karchmer and D.S. Stellenberg. "Using assignment decision diagrams with control nodes for sequential review during behavioral simulation", February 24 2004. US Patent 6,697,773.

[9] S.K.S. Hari, V.V.R. Konda, V. Kamakoti, V.M. Vedula, and K.S. Maneperambil. "Automatic Constraint Based Test Generation for Behavioral HDL Models". *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(4):408–421, April 2008.