High Level Synthesis Using Assignment Decision Diagrams

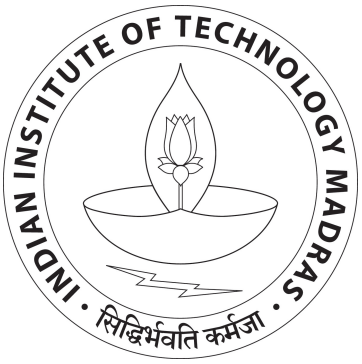
A Project Report

submitted by

B N AVINASH VARMA (EE10B022)

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

BACHELOR OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

11 May 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **High Level Synthesis Using Assignment Decision Diagrams**, submitted by **B N Avinash Varma**, to the Indian Institute of Technology Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work carried out by him under my 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. S. Srinivasan
Research Co-Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

Firstly I would like to thank Dr. V. Kamakoti for guiding me through this project. Without his invaluable support and constant encouragement this wouldn't have been possible.

I would also like to thank Vikas Chauhan and M Pavankumar for their support and for working together with me to bring this project to fruition. I would also like to thank all the members of RISE lab especially Neel for helping and guiding us throughout the project.

I would also like to thank Dr. S. Srinivasan for his guidance and support. I would also like to acknowledge both the Computer Science and the Electrical Engineering Department for giving me this oppurtunity.

I would also like to thank all those who have contributed to making my experience at IITM a memorable one, and all the friends that I have made over these four years. I have learnt more from many of them in these 4 years.

I would like to thank my family for being incredibly supportive throughout and for their constant encouragement and unconditional love.
Avinash.

ABSTRACT

There are numerous problems when it comes hardware design. So in order to solve and optimize these problems we need a high level synthesis of the given hardware design. We are also motivated to develop an end to end solution that can generate gate-level netlist directly from the given hardware design. The concept of using Assignment Decision Diagrams for this high level synthesis of hardware designs provides us with two major capabilities that are not offered by traditional representations, which are, the minimization of syntactic variances and the models for estimating layout quality metrics during synthesis. In addition the diagram also simplifies many tasks such as allocation and scheduling. So we have tried to obtain this higher level of abstraction for hardware written in Chisel which is a Hardware Construction Language designed by U C Berkeley. The language itself is an embedded language based on Scala. The construction of the internal graph representation in Chisel is done using runtime reflection. In order to obtain a higher level design we have leveraged this capability by writing a Scala compiler plugin that goes through the Scala AST and modifies the AST such that the files generated by Chisel have ADD modules which represent the higher level synthesis of the design. We have managed to do this for a few of the main conditional constructs in Chisel.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

AST	Abstract Syntax Tree
ADD	Assignment Decision Diagram
DSL	Domain Specific Language
LUT	Look Up Table
UCB	University of California, Berkeley

CHAPTER 1

Introduction

There has been a tremendous growth in technology in the past few years and orders of magnitude increase in the complexity and the size of hardware designs. This has been primarily driven due to steady growth in the manufacturing process of the transistors which has led to an exponential increase in the transistor count and the performance of the devices. There has also been a great increase in the development of devices that consume lower power and are more efficient.

There are many tools which cater to various stages of a hardware design to generate the final gate-level netlist. We are trying to develop and end to end solution that can generate this directly from a given hardware design. Also there are many other problems that arise due to the increasing complexity of these designs. One more major problem that arises due to this is the problem of desing variances based on syntactic variances in the hardware descriptions. Another problem the issue of testing these complex designs. The aim of this project is try to solve these problems by generating a higher synthesis level of the given hardware description using something called the Assignment Decision Diagrams or ADD's [1] and a new Hardware Construction Language (which is an embedded DSL in Scala) called Chisel which is being developed by the UCB.

Rest of this report is organized as follows. Chapter 2 gives an overview of Scala language and the structure of various nodes in it's AST. Chapter 3 makes the

case for why ADD's were used as for the higher level synthesis of the Hardware Design. Chapter 4 gives a detailed description of on the implementation of our method on some of the constructs in Chisel. Chapter 5 gives a summary of entire report.

CHAPTER 2

Scala Syntax Trees

Scala is an acronym for Scalable Language. The language is scalable as a result of careful integration between the object-oriented and functional language concepts. Some of the advantages of Scala are that it is object-oriented, functional, seamless Java interop and the existence of functions as objects. Scala is a very powerful language with features we feel are important for building circuit generators, is specifically developed as a base for domain-specific languages, compiles to the JVM, has a large set of development tools and IDEs, and has a fairly large and growing user community

There is very little documentation about the Scala Syntax Trees and how to build them. The figure 2.1 below shows the structure of the Syntax Trees and Type trees in Scala and their hierarchy. The best way to learn about them is through printing out the syntax trees for various programs using some compiler flags while compiling and by looking at the various nodes in the tree using a traverser or transformer compiler plugin. Some of the most important nodes in the Syntax Tree that are very useful in the generation of ADDs are listed below. For more information about the Trees you can go through the Scala compiler source code or through the files in scala.tools.nsc.ast package especially through NodePrinters.scala or Trees.scala files.

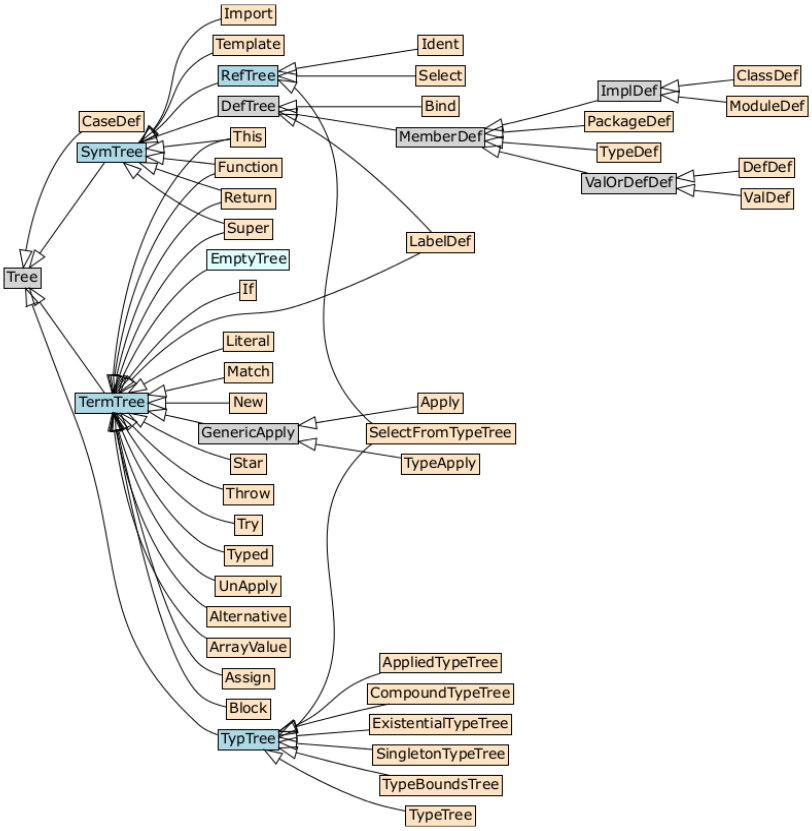


Figure 2.1: Structures of Boltzmann Machine and Restricted Boltzmann Machine

2.1 ClassDef

This is one of the most important nodes in the AST. It is used to define classes in the source code. It has four parameters or children, they are mods, name, tparams, impl. The mods parameter contains information about the scope of the class like private, public etc, it also contains information about the location of the class in the source code file. The parameter name contains the name of the class while tparams contains information about the type of the class. The impl node is of the type Template node which contains information about the parents of the Class and the body of the class.

2.2 ValDef

This node is used to represent all the declarations of variables both mutable and immutable. Once again this node consists of mods, name, tpt and rhs as its fields or children nodes. The mods parameter signifies whether this node is mutable or immutable (i.e val or var in Scala) and also the scope and location of the node in the source code. The parameter name consists of the name of the variable described and tpt is either assigned the inferred type of the variable or meant as a place holder for the type when it is inferred so as to be bound to the node. The rhs parameter consists of the value that is being assigned to the variable.

2.3 Apply

This node is used to represent functions in the Scala Syntax Tree and is one of the most important nodes in our project because Chisel has defined its own functions for all the operations. It also has its own datatypes and all the functions and assignments done in Chisel appear as Apply nodes in the AST. It contains two children nodes fun and args. The fun nodes consists of the function node which is to be applied on the args node which consists of a list of arguments.

2.4 Select

The Select is also very important as it represents the "." Operator and some of the symbols used for functions in the Scala AST. If the name of variable contains a "." Then the Select node has two children qual and name to denote the part that comes before the "." and name to denote the part that come after ".". Also various

operators and symbols like “==”, “∨”, “&&” etc are represented using this node in the AST.

2.5 Block

The block node is usually used to represent blocks of code in the AST. This contains two children stats and expr. The expr contains the node that has been assigned the last in the block of code i.e usually the last Apply or ValDef or ClassDef node in the block. The stats node contains all the other nodes and assignments in that block of code.

Some other very important nodes in the AST are the Ident, Literal and Constant nodes which are used to describe the string names or values passed on for the names of the various classes, methods, values or parameters. Ident and Constant nodes take just the parameter as a String and the Literal node takes the Constant node as a parameter to generate a Literal Node which can then may be passed as argument in the args for an Apply node. Some of the representations of the various nodes in AST is as follows.

+	Ident(“$plus”)
-	Ident(”$minus”)
*	Ident(”$times”)
/	Ident(”$div”)
+=	Ident(”$plus$eq”)
&&	Ident(“ampamp”)

||	Ident(“barbar”)
~	Ident(“unary_$tilde”)
val x = 10	ValDef(Modifiers(0),newTermName(”x”),TypeTree(),Literal(Constant(“10”)))
io.a	Select(Ident(”io”),newTermName(”a”))

CHAPTER 3

Assignment Decision Diagrams

In the following chapter we talk about the importance of ADD's and how they help in the high level synthesis of hardware designs.

3.1 Introduction

The primary objective in deriving ADD is to define a representation that is capable of encapsulating functionality of the described hardware in a unique, precise and simple manner. These three are very important because of the following reasons

- The uniqueness of the representation will allow synthesis tools to be independent of syntactic variances that are present in the input description. Hence, the ADD has to be able to depict the most parallel representation of a given description in order to satisfy the uniqueness property.
- In addition to being unique, the representation we are seeking should consists of parts that reflect semantics of the description instead of syntactic constructs. Each constituent of the presentation should have no direct relationship with language constructs. We refer to this property as the preciseness of the representation.
- A simple representation is one that consists of a few number of different object Types and relationship between each object type. Such representation

can simplify synthesis algorithms because the algorithms have to manage small number of objects. Since most of the synthesis algorithms are top ology-graph based, the representation for a synthesis system has to be a form of topology graph. Thus, a simple representation is, ideally, a graph that consists of small number of different types of nodes and edges.

The figure below show a sample ADD for a FSMD model.

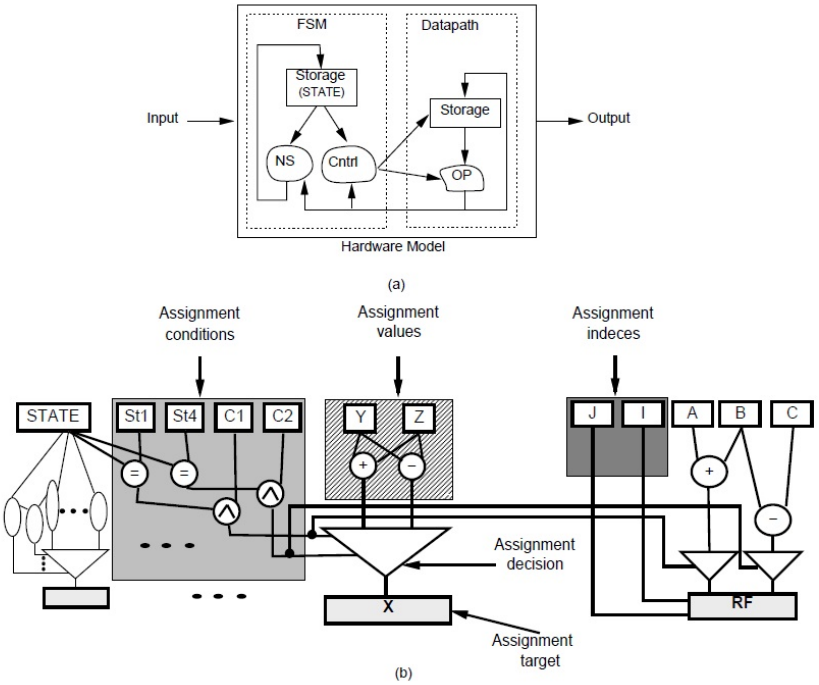


Figure 3.1: The Assignment Decision Diagram: a) FSMD (Finite State Machine Datapath) model, b) the ADD

3.2 Advantages of ADD's

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. In other words, each language construct in the

description is realized with a particular topology of nodes in the representation. Due to the one-to-one correspondence that exist between the constructs of the input descriptions and the schema for the internal representation, these compilers produce different representations for different descriptions. The internal representations of two given descriptions could be far different even if the descriptions are semantically equivalent.

Graphs obtained from the compiler are used by high-level synthesis tasks. Hence, majority of synthesis algorithms are topological-graph based. These algorithms produce results that are generally depended on topology of the graph. Meaning, the algorithms would produce different results for graphs with different topology, even though those graphs have the same semantics. And since the compiler produces different graph topologies for different descriptions, as the result, synthesis tasks would produce different hardware for each of the topology, as illustrated in Figure below.

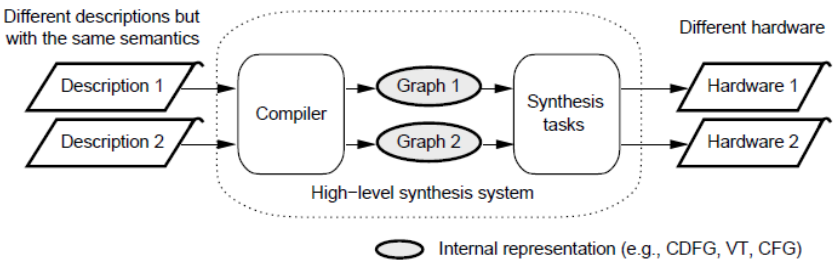


Figure 3.2: A general High Level Synthesis approach

ADD can reduce the impact of syntactic variances without unnecessarily increasing the complexity of synthesis tasks by improving the internal representation and modifying the compiling scheme.

It is capable of representing different descriptions that have the same semantics in one unique topology. There are many ways of representing a given description starting from the most sequential, which is inherent from the description, to the most parallel representation. ADDs give the most parallel representation to be the unique representation because it does not contain implicit sequentiality that are found in the description.

Once we have the ADD we develop a compilation scheme from the input description into the new representation. The results obtained this kind of transformation are consistent and dont depend on the ordering or grouping of conditional branches and/or computations.

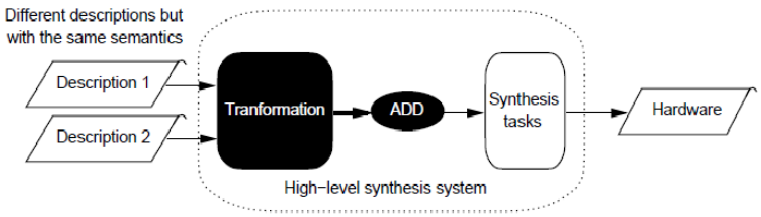


Figure 3.3: An overview of the compilation scheme using ADD approach

3.3 ADD representation of the Chisel constructs

This section looks at the ADD representation of two of the main Chisel conditional constructs. When-Eleswhen This construct is used in the behavioral description to produce sequential condition-branching effect. The when part is accompanied by a condition that if evaluates to true will cause the execution of actions in the

When part, otherwise actions in the elsewhen part are executed if the condition accompanying the elsewhen construct evaluates to true. For example, consider the following sample of code,

```
1 when (C == 0010) {
2     A := B + D
3 }.elsewhen (C != 0010) {
4     A := B - D
5 }
```

The corresponding ADD representation of this piece of code is as follows.

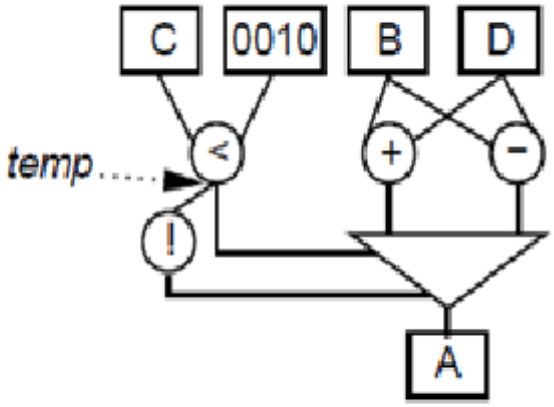


Figure 3.4: An example of the ADD representation of the when-elsewhen construct

Switch Similar to the when-elsewhen construct, the Switch construct provides multiway branching capability to the sequential description. Actions in each branch is executed if its companion evaluates to true.

The parallel representation of the Switch construct requires simultaneous evaluation of conditions and execution of operations in each branches of the Switch. For example consider the following switch statement.

```
1 switch(C + E)
2 {
3       is(0001) { A = 0100; }
4       is(0101) { A = 1111; }
5       is(1001) { A = 0110; }
6 }
```

The corresponding ADD representation of this piece of code is as follows.

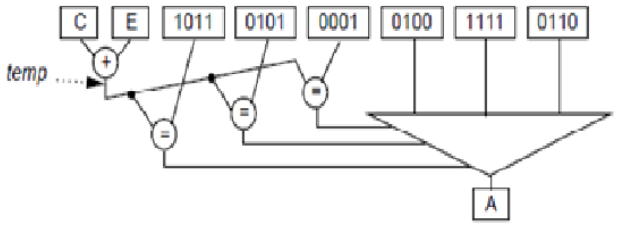


Figure 3.5: An example of the ADD representation of the switch construct

CHAPTER 4

Scala Compiler Plugin

This chapter describes the process of converting the given hardware description to a higher representation level using ADD's

4.1 Introduction

To overcome the challenges in complexity and the increasing size of hardware designs we chose the Chisel Hardware Construction Language which is being developed by U C Berkeley which was primarily designed with goal of Hardware Construction rather than other traditional Hardware Design Languages which primarily emerged for testing the Hardware Designs. Because the semantics of these languages are based around simulation, synthesizable designs must be inferred from a subset of the language, complicating tool development and designer education.

Chisel supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. It has the following advantages.

- Hardware construction language (not C to Gates)
- Embedded in the Scala programming language
- Algebraic construction and wiring
- Abstract data types and interfaces

- Bulk connections
- Hierarchical + object oriented + functional construction
- Highly parameterizable using metaprogramming in Scala
- Supports layering of domain specific languages
- Sizeable standard library including floating-point units
- Multiple clock domains
- Generates high-speed C++-based cycle-accurate software simulator
- Generates low-level Verilog designed to pass on to standard ASIC or FPGA tools

In Chisel all the Datatypes are defined separately from Scala's internal Datatypes. The functions operating on these Datatypes are also defined by overriding the default functions. They also use their own constructs for every definition. As a result of this we observed that all their methods were just functions that were defined separately. The language itself works at runtime using reflection to generate the intermediate graph which is then used to create the Verilog or C++ descriptions of the hardware. But we didn't want to lose any of the behavioural information regarding the hardware description and hence we wrote a compiler plugin that goes through the AST to generate ADD representations of the code.

Scala has a particular format for writing a compiler plugin which is described in their site on how to create and initiate a compiler phase and which stage the plugin can run after by setting some of the parameters in the Singleton created by extending the PluginComponent class described internally in Scala. Also since

we are trying to transform the AST we need to extend this singleton with the trait Transformer. This is very important as if we dont then the Syntax Tree returned by us will not be actually replaced with the original and given to the remaining phases of the compiler. Also one more thing that seems to happen is that no matter where you set your compiler phase to run either after the namer phase (2nd phase in the compiler which resolves names, attaches symbols to named trees), the packageobjects phase (3rd phase in the compiler which loads package objects) and the typer phase (4th phase in the compiler that types the trees i.e infer and determines the types of all the objects present in the AST) the compiler phase only seems to run after the typer phase. The different phases of the compiler can be seen by using the Xshow-phases Ydebug flags while compilation of the code.

In our case we decided to run the compiler after the parser phase where we dont have to provide information about the types of the objects yet and providing just placeholders for them in the code is enough. Also the tree generated at this stage is sufficient in determining all the constructs in Chisel because Chisel has its own Datatypes and all the operations used by this Datatype are just separately defined functions. Apart from that all the language constructs of Chisel are also defined as separate functions in the Chisel library. Hence information about all the Datatypes and Constructs of the language can be inferred from the tree generated at this point of time as they it doesnt use any first order types or functions used by the Scala language they just appear as individual functions in the AST.

We leverage this to generate a list of all the conditional assignments for various nodes (i.e the output nodes) in the AST.

4.2 Traversing the AST

The singleton which extends Transform trait in the plugin overrides the transform function of the trait. This function is called recursively for all the nodes in the AST by the compiler. Inside this function we check if the node is of the type ClassDef as all modules in Chisel are Scala classes which extend the parent class Module. So we also check that the parents of this node contains Module. Once we have the right node we go through all the children in the body of the ClassDef child Template node (this consists of the body of the node and also the parents). Whenever we come across one of the Chisel constructs of when-elsewhen, otherwise and the switch constructs in the AST which exist as Apply nodes in the AST we look at the condition and the body of these nodes. Also we start storing the variables that are being assigned at this level in a hash table by checking if the Select node which is the child node of the apply node which we are looking into consists of the ":=" operator in the name child.

Since the when-elsewhen-otherwise construct is also appears entirely as one single Apply node we travel recursively through the Apply nodes that are present in the children of the template block of the ClassDef node. The otherwise node is in the name part of the Select node which is in turn a part of the fun block of the Apply node that carries out the function of the otherwise block. The args part of that Apply node contain the body of the otherwise node i.e assignments that are made inside this block.

Similarly when it comes to the elsewhen construct, it also appears in the name part of the Select node, which in turn is a part of the fun node of the parent Apply node which in turn is again part of the fun node of another parent Apply node. The grandparent Apply node contains the args parameter that contains the actual body of the elsewhen construct i.e the assignments that are made if this condition actually holds. The parent Apply block contains the args parameter that contains the condition that this elsewhen block is evaluating against.

The when construct, appears in the fun part of the parent Apply block as an Ident block in the AST. The parent Apply is again entirely part of the fun block of another Apply block. The grandparent Apply node of the when construct similar to the elsewhen construct and contains the body of the actual assignments of that the when block runs of correct evaluation. Also the part Apply block is similar to the elsewhen case in that it's args also contain the condition which the when block is evaluating against. This is a sample of the Apply node which is a child node of the ClassDef node that consists of one when, two elsewhen and the otherwise constructs.

```
1 Apply(
2     when(io.opcode.$eq$eq$eq(UInt(0)))(io.output.$colon$eq(io.a.$plus(
          io.b))).elsewhen(io.opcode.$eq$eq$eq(UInt(1)).unary_$tilde)(io.
          output.$colon$eq(io.b.$minus(io.b))).elsewhen(io.opcode.
          $eq$eq$eq(UInt(2)))(io.output.$colon$eq(io.a))."otherwise"
3     Apply(
4           "io"."output"."$colon$eq"
5           "io"."b"
6          )
7     )
```

Since these constructs are not present as individual nodes and exist together we recursively iterate through them and at the same time store the conditions and the arguments these nodes are being assigned in these blocks and add the additional conditions and assignments these nodes have.

To account for the case of nested constructs we have to store the conditions at least up to that level. We do this by passing the parentCondition variable recursively while parsing the graph. Also we need to pass the new condition assigned only to the args of the grandparent nodes in the case of when and elsewhen nodes and the parent node in the case of the otherwise node.

4.3 Tranforming the AST

Once we have all the available nodes in the that have been assigned values inside the class using ":=" which is the operator used specifically in Chisel to assign values we try to transform the AST.

Firstly the template calls the pretransform function in the overridden transform function of the internal Scala Transformer trait, then the superClass implementation and finally the postTransform function on all the tree nodes generated for the given source file. So when we have a new class definition that's actually a module we store a list of conditions and assignments for those conditions for various variables in the code. Whatever Tree the the posttransform takes as input will be substituted with the output Tree that it actually returns. This is how the transformers class works.

The scala AST is immutable and so change the AST we need to use one of the internal functions provided in the compiler called treeCopy.

4.3.1 treeCopy

This is a very important function in the compiler that helps us transform the AST's of the source code which are immutable. The way it works depends on the node that you want to replace. For instance in the case of a ClassDef node then we call the treeCopy.ClassDef function which takes in 5 arguments in this case. The first argument in the old ClassDef node you're trying to replace. The remaining four arguments are the Children of the ClassDef node (i.e mods, name, tparams, impl) that the new ClassDef node is made up of.

Similarly if you want to replace the ValDef then you call the treeCopy.ValDef funtion which similar the above function takes in 5 arguments. The first argument is the original node that you want to replace, and the remaining are the modifiers, the name, the inferred type of the class or a placeholder for it and the rhs value

that's actually being assigned to the node.

4.3.2 newTypeName

Also once we have the list of conditions and assignments for the various variables in the code we use the treeCopy functions and define the ValDef node for the initialization of the corresponding ADD modules from the ADD library for these Chisel constructs written in Scala by Pavan. But it's very important to note that the scala AST is not being able to recognize the Class name if just passed as an Ident or newTerm in the AST. The compiler seems to think that this is just another variable in the AST and throws up an error when it fails to find it. So we must use the newTypeName function defined inside the Scala compiler to initialize classes defined outside. The following is a sample implementation of this function.

```
1 class A {
2     println("Hello");
3 }
4 object Avi {
5     def main(args: Array[String]) {
6         var t = ValDef(Modifiers(0),newTermName("test"),TypeTree(),
              Apply(New(Ident(newTypeName("A"))),newTermName("<init>")));
7     }
8 }
```

We can pass empty modifiers since we don't want to set anything regarding the scope and the default scope assigned is public, The function newTermName is also very important in describing TermSymbols in the Scala AST which how all the other symbols which are not Classes, methods, keywords or types are described.

So we use it to create a termSymbol for the name of the value in AST so that it can recognize it as a value and add it to the symbol table for evaluation during the subsequent phases. The tpt value at the parser phase can be empty as the inferred type is attached here in the typer phase of the compiler. Hence we just initialize an emty TypeTree(). The rhs value in this case is a new instance of the class A. Hence we use the New node which describes the new keyword in the AST and as you can see since the class name is given to the newTypeName so that the compiler can correctly map that it's a type or class in the subsequent phases and not just another variable.

So once we have the ADD library that has the equivalent representation of these constructs to their respective modules we can initialize the right classes by creating appropriate nodes for them in the AST. Also we can then set the conditions and arguments as inputs and outputs for them by creating apply nodes which make use of the ":=" operator used in Scala.

CHAPTER 5

Summary, Conclusions and Future Work

We have tried to build an end to end solution that generates a gate level netlist from the given hardware description and uses high level synthesis using Assignment Decision Diagrams so as to overcome some of the challenges posed due to the increasing complexity and variety of designs. The project is still in a early stage. We have currently successfully maped the when, elsewhen, otherwise and switch constructs in the Chisel language which are the major conditional constructs.

The plugin was tested on sample Chisel codes which generated Verilog both before and after transformation. The functional integrity of the code was verified using Formal Pro.

Once we have an equivalent graph of all the constructs in the Chisel language like support for “for” and the “while” loops in the ADD library then we can directly generate an LUT representation of these various structures in the library and the various operators used in Chisel. Then these can be used to generate a gate level netlist for the entire representation. This is an implementation of the method described in the Section 3.2. Also you can optimize the logic [2] and generate automated test vectors for this representation mentioned in [3] and in [4].

CHAPTER 6

References

1. Viraphol Chaiyakul, Daniel D. Gajski and Loganath Ramachandran "High Level Transformations for Minimizing Syntactic Variances"in 30th conference on Design Automation, 1993.

2. Kuang-Chien Chen, Jason Cong, Yuzheng Ding, Andrew B. Kahng, Peter Trajmar "DAG-MAP: Graph-Based FPGA Technology Mapping for Delay Optimization" in IEEE Journal of Design and Test, 1992.

3. Liang Zhang, Michael Hsiao, Indradeep Gosh "Automatic Design Validation Framework for HDL Descriptions via RTL ATPG" in IEEE Test Symposium , 2003.

4. Indradeep Gosh, Masahiro Fujita, "Automatic Test Pattern Generation for Functional Register-Transfer Level Circuits Using Assignment Decision Diagrams" in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on (Volume:20, Issue: 3) , Mar 2001.

5. Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas AviÅienis, John Wawrzynek, Krste Asanovic "ImageNetChisel: Constructing Hardware in a Scala Embedded Language" University of California, Berkeley, 2012