High Level Synthesis Using Assignment Decision Diagrams

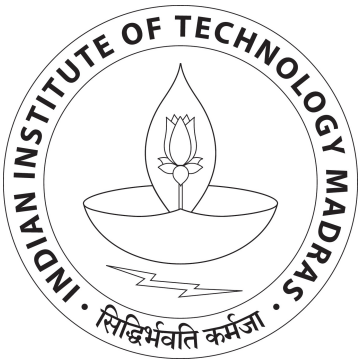
A Project Report

submitted by

PAVANKUMAR REDDY MUDDIREDDY (EE10B020)

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

13 May 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **High Level Synthesis Using Assignment Decision Diagrams**, submitted by **Pavankumar Reddy Muddireddy**, 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 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 Dr. S. Srinivasan for providing necessary help and support through the project.

I would also like to thank Vikas Chauhan and B N Avinash Varma 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 everyone in the institute who have contributed to making my experience at this great institute a memorable and wonderful one, and to all my friends in my department, wing and elsewhere from whom I learnt a great deal during the course of my undergraduate years.

I would like to thank my family for being incredibly supportive throughout and for their constant encouragement and unconditional love. I am eternally gratefully to them for the same.

ABSTRACT

When it comes to hardware design, various approaches are adopted and each approach brings with it various problems. In this project, we are try to optimize the design at higher level of abstraction. We take the concept of Assignment Decision Diagrams for high level synthesis to provide an end to end solution that can generate gate-level netlist directly from a given hardware description described in "Chisel Hardware Construction Language". Chisel, which is an embedded DSL in Scala, being a higher level language and designed specifically for Hardware Construction helps us put more focus on 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. The language itself is an embedded language based on Scala. ADD leverages the concept of digital design being a collection of conditional assignments to output ports and wires. I modified the Chisel compiler by adding a backend to it which generates Verilog files after the description has been synthesized into ADDs.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

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

CHAPTER 1

Introduction

Technology that is available today enables companies to build large and complex systems with millions or even billions of transistors on a single chip. In the recent years, people started using logic synthesis methods to design process i.e we see an evolution from capture and simulate to describe and synthesize methodology. The advantage with this method is that we can give an input description in purely behavioral form devoid of implementation details and designers can apply the describe and synthesize methodology using various levels of abstraction - gate level, FSM level or RTL level with flow charts and dataflow graphs.

Using high-level design, designers can achieve high levels of productivity by using the automation and higher levels of abstraction provided by high-level synthesis.

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 hardware from a given description in Chisel (explained below). This optimal hardware is to be syntactically independent and is the most parallel representation.

We try to achieve what was stated above by using structures called Assignment Decision Diagrams (ADDs)[1] with input description in Chisel which is a Hardware Construction Language based on Scala being developed by UCB.

Rest of this report is organized as follows. Chapter 2 gives an overview of Chisel, its various constructs, some of the internal working of the Chisel compiler and internal working of a custom backend. 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 little overview of Scala compiler plugin and the changes made to it so as to get a high level synthesis. Chapter 5 gives a summary of entire report.

CHAPTER 2

Chisel

This chapter describes the Chisel programming language and the reason for choosing Chisel as our HDL.

2.1 Introduction

The traditional hardware-description languages were originally developed as hardware simulation languages, and were only later adopted as a basis for hardware synthesis. 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. Also since the languages Verilog and VHDL were developed a long time ago they also lack powerful abstraction facilities that are common in modern software languages which leads to low designer productivity by making it difficult to reuse components.

Chisel being a higher level and modern language based on Scala facilitates the composition of highly parameterized module generators which is very crucial if you want to do an extensive and thorough design-space exploration. Althoug some of recent improvements like SystemVerilog address some of these issues they still lack many of the powerful programming language feautures. Another advantage that Chisel has over languages like Esterel, DIL or Bluespec is that these

languages follow a pattern encoded programming model that is suitable and very powerful when the design that is being developed matches the pattern whereas the higher computation level of Chisel helps us avoid these problems.

Chisel was developed from the beginning with the goal of being simple platform that provides modern programming language features for accurately specifying low-level hardware blocks, but which can then be readily extended to capture many useful high-level hardware design patterns. By using a flexible platform, each module in a project can employ whichever design pattern best its that design, and designers can freely combine multiple modules regardless of their programming model. Hence Chisel was the HDL of our choice when we started this project.

2.2 Chisel Datatypes

Any hardware design in Chisel is ultimately represented by a graph of node objects. The Chisel type system is maintained separately from the underlying Scala type system, and so type nodes are interspersed between raw nodes to allow Chisel to check and respond to Chisel types. Chisel type nodes are erased before the hardware design is translated into C++ or Verilog. The figure below shows the built-in Chisel DataType hierarchy. There are built in scalar types like Bits, Bool, Int, and UInt and then there are some built in aggregate types like Bundle and Vec which allow you to expand the set of Chisel Datatypes with other collection types. The datatype nodes are added to the node graph that is generated while reflecting the source file. The Type nodes are attached to Op or Lit nodes which hold

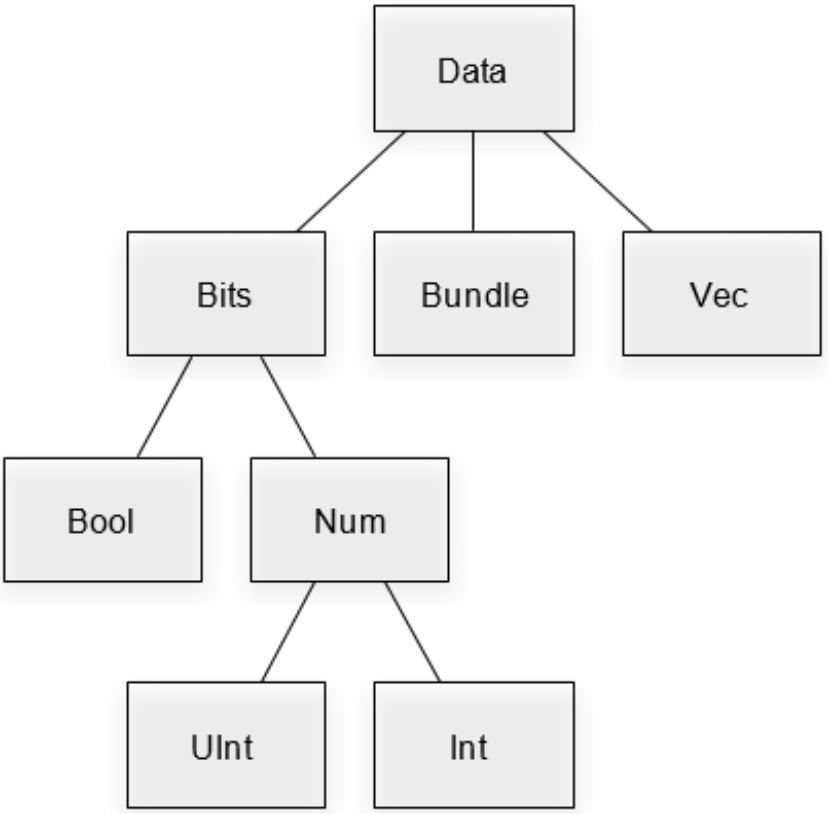


Figure 2.1: Chisel DataType Hierarchy

information about the names of the variables or the operations being performed on them. The below diagram shows a sample of these graphs.

2.3 Chisel Internals

In this section we delve deeper into how the Chisel programming language actually works. Some of the main components of the Chisel compiler are described below

2.3.1 Node.scala

This the main scala class that is used to model all the types in Chisel. The internal graph that Chisel using for representation of the source code is built from the

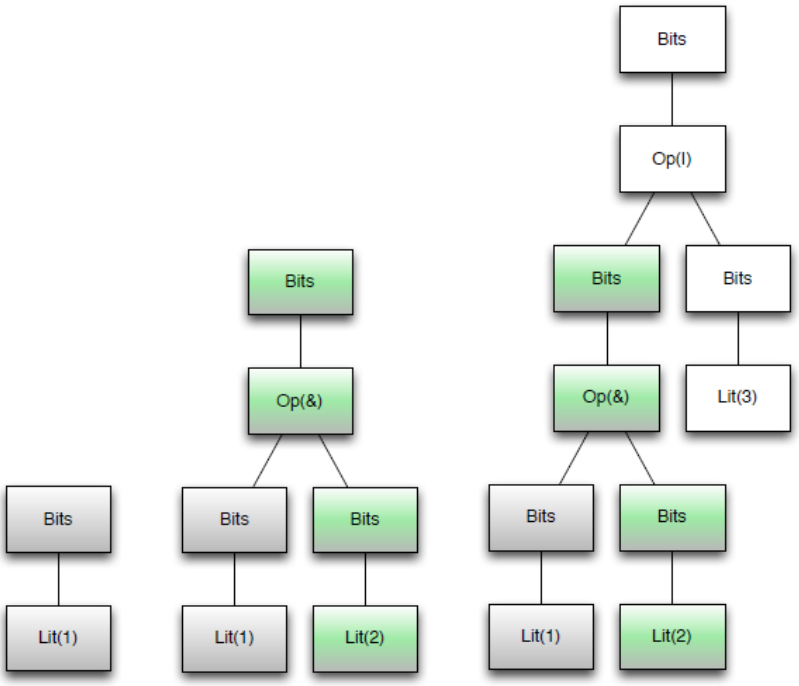


Figure 2.2: Chisel Op/Lit graphs constructed with algebraic expressions showing the insertion of type nodes

instances of this Node class. The figure below shows the hierarchy of classes in Chisel.

2.3.2 Module.scala

This is one of the most important parts of the compiler. In the hardware description all the hardware components extend this singleton. The compiler itself uses reflection on this object and finds the methods described in the source. Since the Chisel Type system is different from that of Scala and all the functions are defined separately on these types (separate from original scala functionality) all the Chisel constructs defined inside this module at are evaluated as methods at runtime which is extracted using reflection by the Chisel compiler.

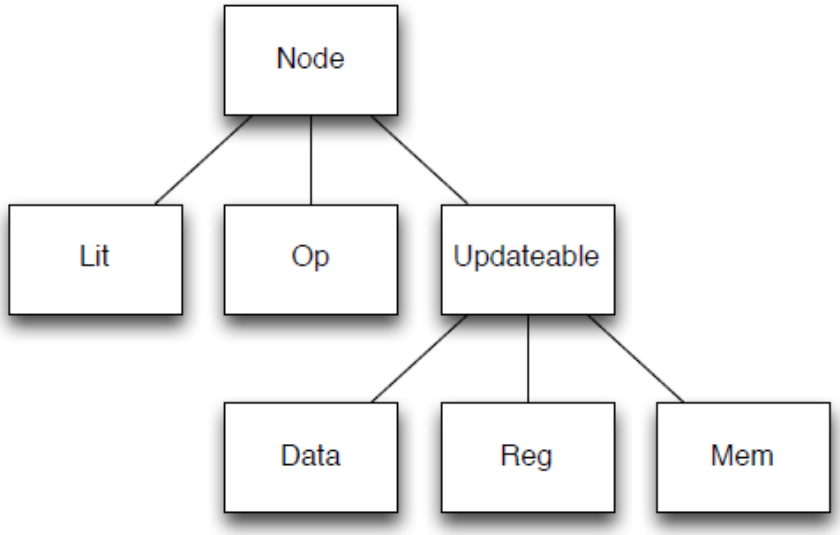


Figure 2.3: Chisel Node Hierarchy

2.3.3 Backend.scala

This is the core of the compiler where the reflection on the Module or in this case it's subclasses the modules described in the hardware description. The reflection happens in the elaborate function of this class. The function is not called directly but by one of it's subclasses because every Module has a backend parameter that contains an instance of this class or it's subclasses. This backend parameter is set based on the command line arguments passed by the user. This file also provides the necessary functions for extending the capabilities of the compiler by writing new backends that are children of this class.

2.3.4 Verilog.scala

This is the file that generates the verilog file from the intermediate Chisel graph that is built. It contains some valuable functions to convert the graphical representation to a verilog file. Much of the backend written by us was done by understanding

and using the functions written in this file. It's elaborate function can be used to access the Chisel intermediate graph.

Some other important files are the Datatype files like Data.scala, Bits.scala etc., which can be gone through to know more about the kind of functions you can implement or to extend the functionality of these datatypes. Also the file hcl.scala is important for adding new backends as this the file that contains the definition of ChiselMain function which is used to instantiate the chisel modules in the scala source. You should add your custom backend here and customize the command prompt options to run Chisel. Also any other changes to runtime arguments can be done in this file.

2.4 Custom Backend

We implemented a custom backend to the compiler by adding a new backend class that extend the Backend class in the Chisel compiler. In this backend we went through the Chisel graph i.e the various nodes that is being generated by Chisel by going through the tree in reverse from the outputs to the inputs. The functions getRoots defined in Module can be used to get the roots of the Module i.e the output nodes that are being assigned. The variables that are being assigned in these nodes were stored along with the conditions that were being evaluated for the assignments. The conditional arguments appeared in the graph as MUX nodes. Once we had all the conditional and behavioural logic from the internal graph of Chisel we generated the new verilog file that modeled the description

in terms of ADD's. A separate ADD.v file was written that contains the verilog representation of the ADD modules. The generated verilog files with and without the new backend files were verified later.

CHAPTER 3

Assignment Decision Diagrams

In this chapter, we see about the workings of an ADD and advantages of the structure and how it contributes to the abstraction in the high-level-synthesis. Some of the verbatim in this chapter is taken from the paper by D.Gajski.

3.1 Introduction

ADD was primarily defined to encapsulate the functionality of a given hardware description in a unique, precise and simple manner. They are important because,

- 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.

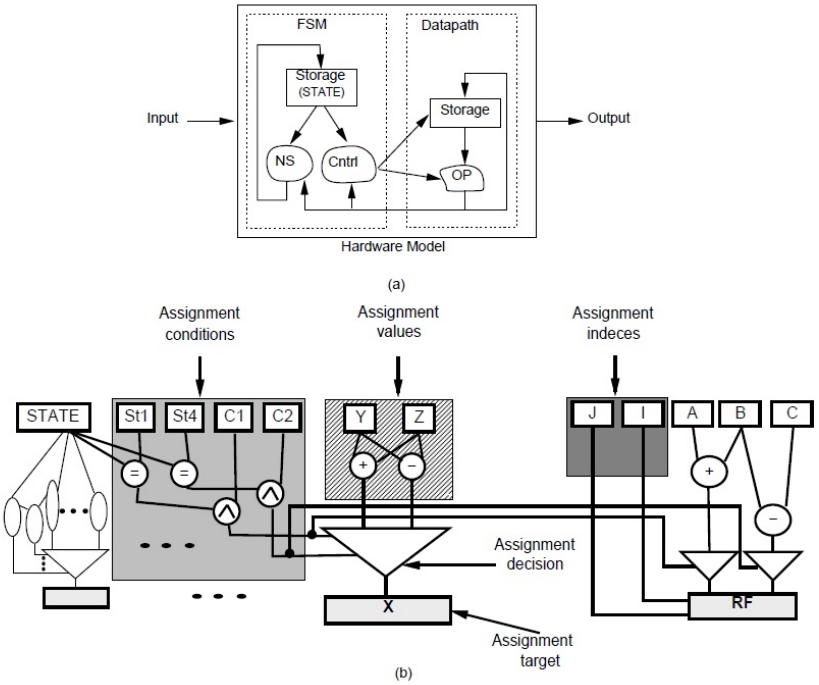


Figure 3.1: The Assignment Decision Diagram: a) FSMD (Finite State Machine Datapath) model, b) the ADD

3.2 Advantages of ADD's

As discussed in the introduction, high-level synthesis converts a given input description with a high level abstraction into an internal representation. All synthesis tasks works from this description.There are several types of internal representation. The most convenient type is the one that matches the problem most closely.

For example, for a digital filter, which repeatedly performs a series of operations on an infinite input data stream, we want to represent the data, the arithmetic operations, and the read and write dependencies that define the order of execution. A dataflow graph (DFG) is the best way to do this. But the problem with various kinds of internal representation is that usually there would be one to one mapping between various kinds of input constructs and different nodes of the internal representation.

This would lead to a situation where two behaviourally (semantically) similar descriptions with minor syntactically differences would lead to different output hardware. A large number 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. ADD can reduce the impact of syntactic variances

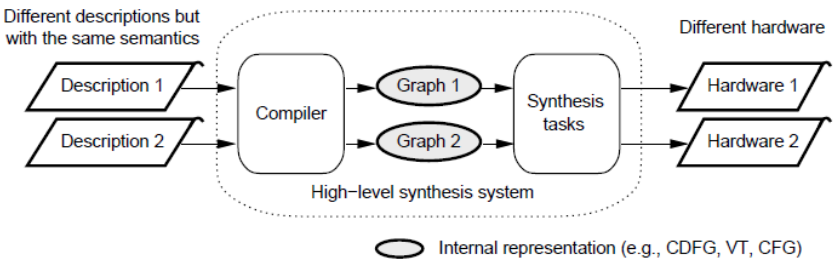


Figure 3.2: A general High Level Synthesis approach

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.

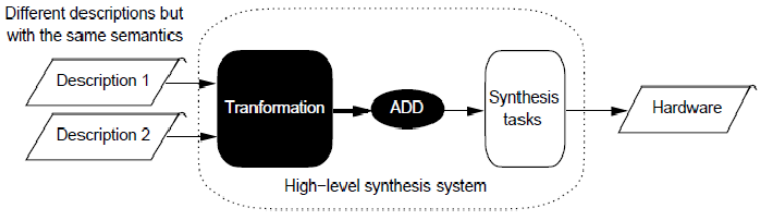


Figure 3.3: An overview of the compilation scheme using ADD approach

3.3 ADD representation of the Chisel constructs

This section looks at the ADD representation of two of the main Chisel conditional constructs.

3.3.1 when-elsewhen

We use when-elsewhen to get if-else functionality i.e conditional branching capabilities. Consider the following sample of code as an example to when-elsewhen

behavior,

```
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.

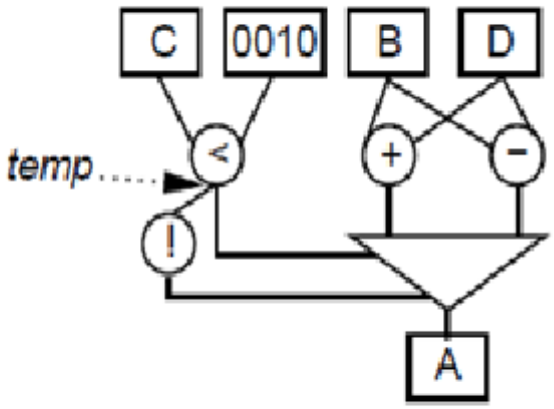


Figure 3.4: An example of the ADD representation of the when-elsewhen construct

3.3.2 switch

Switch construct is similar to switch construct in C. It is an alternative to when-elsewhen construct i.e the Switch construct matches one of the cases provided or goes to default. It provides multiway branching capabilities.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( 1011 ) { A = 0110 ; }
6 }
```

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

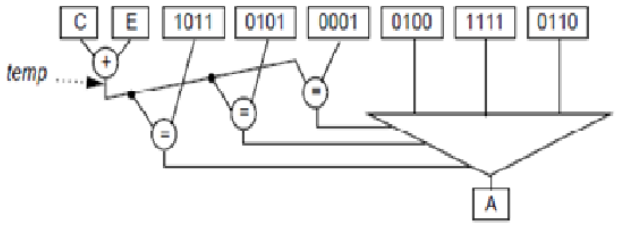


Figure 3.5: An example of the ADD representation of the switch construct

CHAPTER 4

Scala Compiler

In this chapter we disuss the implementation of the various features to the scala compiler plugin that does the high level synthesis of a given hardware description using an Assignment Decision Diagram as described in Chapter 3

4.1 Introduction

The Scala is a very good functional and object oriented programming language. All the functions in Scala are treated objects. The language has many features that are important for the generation of circuits and the language has also been developed with aim of being a base for domain-specific languages.

We are using a compiler plugin to extract the AST that contains the complete hardware description so as to not lose any information regading the behavioural or combinational logic in the description. I used the same compiler plugin that is being developed by Avinash. The AST generated by the scala compiler can be used to synthesize the hardware description using ADD. The scala AST can be analyzed by printing and looking at various nodes that get created using the compiler flags -Xprint:parser -Yshow-trees. The parser denotes the phase after which the compiler runs. This is the first phase of the compiler where just syntactical and static type inference is done. The next three phases the Namer, Packageobjects and

the Typer phase deal with the creating the symbol table and the typing of the trees. The symbol table can be printed using -Yshow-syms.

Some of the important nodes in the AST are ClassDef, ValDef, Apply and Select. These most important nodes in our case are Apply and Select as the Chisel library functions all appear as Apply nodes and the Select node is used by Scala for the "." syntax terms in the source code. So we analyse the Apply node and the Select nodes in the body of the classes that extend module class and then build our ADD representations for these outputs.

4.2 ADD Extension

The ADD library has support only for ADD's of size upto 8 conditions and 8 arguments. So we extended the ADD support for variables with higher order assignments and conditions by reusing the ADD and adding levels of ADD's similar to the way muxes are arranged in the case of higher order computations. The figure below shows this nested arrangement.

4.3 Maintaining Variable Scope after ADD transformation

In Scala, like most other languages, variables are reserved memory locations used to store various values of different types. Based on the type of the variable declared, the compiler allocated memory depending on the type. The type of variable can

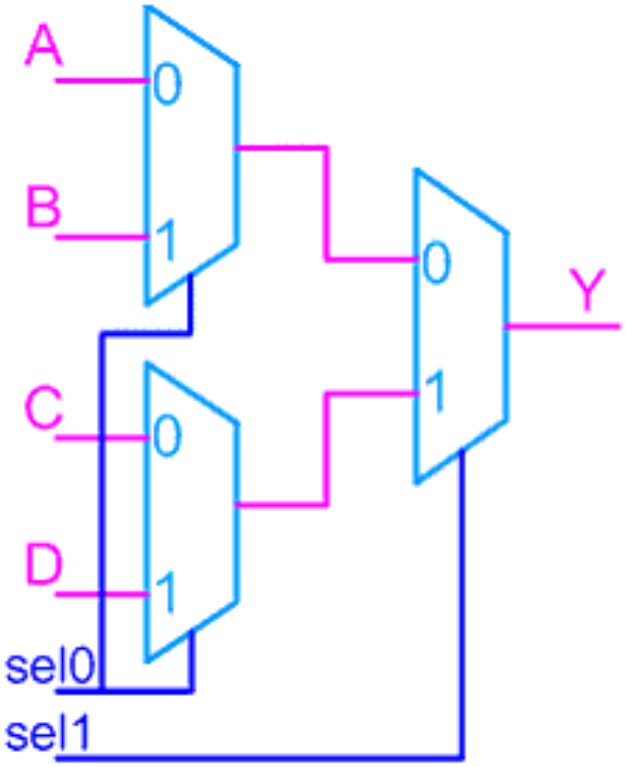


Figure 4.1: Nested ADD's for higher order assignments

either be explicitly declared or the compiler will try to infer the type based on the right hand side of the expression. The different types of data that is available in Scala include but not limited to Integers, Decimals, characters, Strings, etc. All these are objects unlike Java (no primitive types). We use val and var for declaring variables with val for immutable and var for mutable. Further type can be explicitly declared like below,

```
1 var myVar : String = "Sample"
```

Here, myVar is declared using the keyword var. This means that it is a variable that can change value and this is called mutable variable. Following is the syntax to define a variable using val keyword:

```
1 val myVal : String = "Sample"
```

Here, myVal is declared using the keyword val. This means that it is a variable that can not be changed and this is called immutable variable.

```
1 var myVar = 2;        // Type is inferred here
2 val myVal = "Sample"; // Type is inferred here
```

Variable Types: The variable scopes inside Scala is very similar to other languages with 3 main scopes to the variables. These are the following,

- Fields (instance variables):

  These are variables declared as member of the class. These members are accessible only from inside the object i.e they can be accessed by member methods or set by the constructor. Furthermore, these can also be accessible outside the object depending on what access modifiers. Object member variables can, as described above, var or val i.e mutable or immutable.

- Parameter Variables (Method params):

  These are variables passed onto the methods. The scope of these variables are only inside the method. But some variables like objects which are passed by reference than by value can be accessed form outside. These are always implicitly immutable i.e val.

- Local Variables:

  These are variables declared inside a block or method. Blocks in Scala are pieces of code enclosed in curly braces. Each block has return value i.e the the last value mentioned inside the block. If nothing is mentioned as the

return value, Unit (which is like NULL) is returned. Local Vars can only be accessed inside the block unless returned by the block as described above and can be both mutable and immutable i.e both var and val.

Chisel type system is distinct from Scala type system (Chisel uses UInt, Bool etc described above while Scala uses Boolean etc) and variable assignments are basically apply methods in Chisel i.e for example,

```
1 io.input := aValue // This is equivalent to io.input.apply(aValue)
```

Since, Chisel variables are also Scala variables, the scope of the variables is maintained. Now, we shall see how the scope of different kind of variables described can be handled in light of creating the ADDs.

Global variables and fields (like io module declared inside of the module class) i.e variables inside the main module can be accessed everywhere, so that would not be a problem. They are left as they are.

Variables inside methods are not being converted to ADDs yet. So, they too need not be handled.

Local variables as described above are the variables declared inside blocks such as that of methods or when or switch etc. Now, these variables could be potentially be used in assignments to output ports. An example of such a scenario is shown below,

```
1 when (somecondition) {
2   var a = UInt(3)
3   io.output := a
4   when {...}          // further blocks
5 }
```

Now, we can see that a is a local variable to the when block and when the io.output is made into an ADD, the reference to the a is lost since the when block is removed while processing. So, we need to change the scope of this variable by moving it to the main module block. So, to prevent a compile time error from occurring, all the variables with local scope have to shifted out of the conditional blocks. This has been done using a Stack data structure. All the variables in a block are pushed onto stack and are renamed to a unique name to avoid conflicts with other variables of same in other blocks. This array is maintained on the stack until we come out of block while adding additional arrays as we go further blocks inside this block and freeing them as we leave the blocks. When we are in a block we replace all the instances of the variables with the newly uniquely named counterpart for that variable. Now all these arrays while are freed are added to a variable list which is added to the main module block at the end.

So, in the above case while the io.output would become an ADD like

```
1 var a_when_#unique_number = UInt(3)
2 ..........
3 ..........
4 io.output = output_ADD.io.out
```

4.4 Scala Library

The main ADD library is currently implemented in Scala with ADDs ranging from 2 input to 8 input ADDs. Each of the ADD module is implemented using a a series of when blocks. Initially, the library was implemented by passing the classname to the ADD library and instantiating depending on the kind of output ports and input ports and the number of bits for various ports in the ADD module was

parameterized. But this led to some errors when elaboration by Chisel failed in a few instances. So, we changed the io port types to Bits which is the super class of the data types used by Chisel.

A sample ADD block is shown below:

```
class ADD2() extends Module {
  val io = new Bundle {
    val in1  = Bits(INPUT)
    val in2  = Bits(INPUT)
    val c1   = Bool(INPUT)
    val c2   = Bool(INPUT)
    val out  = Bits(OUTPUT)
    val default = Bits(INPUT)
    val isActive = Bool(OUTPUT)
  }

  io.out := io.default;
  io.isActive := Bool(false)
  io.out := io.default

  when (io.c1){
    io.out := io.in1
    io.isActive := Bool(true)
  }
  when (io.c2){
    io.out := io.in2
    io.isActive := Bool(true)
  }
}
```

If the number of inputs exceed 8, we extend it as described above. We use

the isActive flag as to know if this ADD has been activated and we use this as a condition to next stage of ADD in the description of ADD extension above.

CHAPTER 5

Summary, Conclusions and Future Work

In this project we have tried to build one of the phases of an end to end solution which would automate the process of producing optimal hardware from a high level behavioral description of a circuit. The high level description in this project is provided in Chisel which is internally converted to the ADD - Assignment Decision Diagrams - descriptions. This ADD ensures the most parallel representation and hence produce optimal results. We have at this stage implemented the ADDs in scala (as described in the previous section) and would like to change it to LUTs as the next stage. We have successfully managed to convert the given description for conditional constructs in terms of ADDs both using Chisel backend and modifying AST using compiler plugin. We have also checked the behavioral equality with the original design with the help of formal pro (using RapidIO library).
Currently because of the scala implementation of the library, the resulting ADDs are not optimal and resulting in an overhead in transistor count and power which needs to be further reduced in future implementations. Furthermore, loop unrolling is not handled currently and needs to be completed. Most of the ideas in this project are taken from the paper mentioned in Chapter 3 by Dr. D. Gajski. Also you can optimize the logic [3] and generate automated test vectors for this representation mentioned in [4] and in [5]

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. Viraphol Chaiyakul, Daniel D. Gajski "Assignment Decision Diagram for High Level Synthesis", Technical Report 92-103, University of California, Irvine, December 1992

3. 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.

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

5. 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.

6. 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

7. Gajski, Daniel and Dutt, Nikil and Wu, Allen and Lin, Steve, "High Level Synthesis", Kluwer Boston, 1992