School of Computing

National University of Singapore

CS4215 - Programming Language Implementation

DSL (Domain Specific Language) Project

Chisel: Improving Comparison Operators

Final Report

Members:

Name	Student ID	Mail
Huaqiang Wang	A0196556M	wanghuaqiang16@mails.ucas.ac.cn
Fabian Reinisch	A0196971L	E0392413@u.nus.edu

Submission date: 2019-04-26

1. Introduction

Chisel is a hardware construction language which is embedded in Scala. Being that, it supports concepts coming from the Scala language, such as functional programming, object-oriented programming and type inference. Because of these properties, it also enables enables easier circuit generation by providing a higher level of abstraction for the programmer. Finally, it also provides advantages compared over the usage of languages like Verilog and VHDL. As these two languages were developed as hardware simulation languages, many constructs are not suited as good for hardware synthesis as well as some constructs are non-intuitive in how they map to hardware implementations.

In Chisel each variable is assigned a “width”. This property is used to set how many bits are being used to store the data for this variable, therefore specifying the range of the variable. In Figure 1, two unsigned Integer variables are being created, with the first one (in0) having a width of eight bits, and the second one (in1) four bits. The result (out) is automatically inferred to be an 8-Bit Integer, so that it can store the result.

Finally, Chisel also supports a feature called “bundles”. This feature enables the programmer to store different variables in one object (similar to structs in C / classes in Java). In Figure 1, the structure “Adder” is a bundle. Chisel also supports nested bundles (a bundle having multiple bundles as attributes inside of it).

```
class Adder(val w: Int) extends Module
{
  val io = IO(new Bundle {
    val in0 = Input(UInt(8.W)) //8 Bits
    val in1 = Input(UInt(4.W)) //4 Bits
  })
  val io.out := io.in0 + io.in1 //8 Bits
}
```

Figure 1: Chisel Adder Code showing the width of variables

2. Improvements

Possibility of comparison outcomes:
In Chisel, the programmer is able to compare different attributes by using different operators: ==, <, >, etc. Here, a common mistake by beginner programmers can occur when instantiating a variable.

```
val counter = RegInit(0.U)
when(counter > 10.U) {
  //doStuff
}
```

Figure 2: Chisel Adder Code showing the width of variables

in Figure 2, since no width was specified when creating the variable “counter” (one the value zero was specified), its width is automatically set to one bit. Now when executing the “when”-statement because of that restriction, this statement will never evaluate to true, therefore rendering it useless. Our first improvement consists of throwing an exception when compiling code with these properties, which enables the programmer to detect these kind of mistakes easily and correct them.

Comparison of bundles:
When creating bundles in chisel, right now, the programmer has to compare each of the attributes separately, or create a function that is able to compare two different bundle objects. However, it’s not possible to just compare two different objects if these are of a bundle type.

```
class MSInt extends Bundle={
      val sign = Uint(1.W)
      val num = Uint(7.W)
}

val a = new  MSInt()
val b = new  MSInt()
.......
val c := a===b
```

Figure 3: Comparison of bundles (not supported currently in standard Chisel)

Figure 3 shows how a possible comparison of two objects which are of a bundle type could look like. The object in Figure 3 consists of an unsigned Integer (7 Bits) which stored the number itself as well as an unsigned Integer (1 Bit) which is used to store the number sign. The second proposed improvement therefore consists of implementing support for a bundle comparison operator. Here it’s notable, that this operations needs to be implemented in a recursive fashion, as in Chisel, bundles might contain other bundles within itself.

3. Usage

This project provides a new scala/Chisel3 package called pliChiselImprove. To use this package in your Chisel3 project, you need to add the following code in your project.

```
import pliChiselImprove._
```

and add pliSupportLib.scala to your Chisel project.

Both strict comparison operators and bundle compare methods can be called using the new function StrictCompare:

```
StrictCompare(input1: Data, operator: String, input2: Data): Bool
```

Where input1 and input2 should be of Chisel's Data type, which means they could be "wires" or "bundles". All Chisel compare operators are supported: <, >, <=, >=, ===, =/=. You can use this function to build synthesizable codes, which means the code can be convert to real FPGA or ASIC design. You only need to import this package where you need to use strict comparison operators or bundle compare. This package works well with the original Chisel3 so you can use Chisel3's built in compare operator at the same time.

This package is able to throw the following exceptions:

CompareWidthMismatchException	Input data width mismatch.
CompareTypeMismatchException	Input data type mismatch, it could because of the bundle struct of the two inputs are different.
UndefinedStrictCompareException	Other undefined exceptions, usually means there is something goes wrong with the package itself.

Figure 4:

Users can track CompareWidthMismatchException and CompareTypeMismatchException to find the errors in Chisel3 codes.

This package is designed to work with Chisel3.1.1 or newer version. It should be able on all Chisel3 projects but the compatibility with the older version is untested. To start a new Chisel3 project, see https://github.com/freechipsproject/chisel-template.

For more detailed examples, please refer to the test set for this package. You can find the full test set at src/test/scala/pliChiselImprove in our template Chisel3 project.

4. Implementation

4.1 Implementation of Comparison Operators

The function StrictCompare(op1, operator, op2) act as a 'StrictMode' for compare operator. To implement this function ,the key point is to detect the width of the operands. For Chisel 'Bits' type like UINT and SINT, there is a built-in method called getwidth. However, for registers in Chisel, their width, if not explicitly assigned, will be automatically inferred in the middle layer: firrtl. Chisel system is consisted of three layers: scala layer, firrtl layer, and verilog layer. Convert Chisel Scala to Firrtl is quite complicated, and the mechanism of Firrtl is quite different from Chisel Scala. As it is difficult to get the width of the regs, this package will require its user to explicitly assign a width of a register. If the user uses a reg as the input of function StrictCompare() without explicitly assign a width, a Compare Width Mismatch Exception will be thrown out. If the two inputs have appropriate type and the same width, StrictCompare() will return the output Chisel Bool wire.

The rules for basic type comparison operators StrictCompare() (pseudocode):

```
if(the two inputs have comparable type (not bundle) ){
    check the width of the inputs{
        case Known width -> Do compare if their widths are the same, otherwise -> Exception
        case Unknown width -> Force user to assign width / width check manually -> Exception
    }
} else {
    Exception
}
```

4.2 Implementation of Comparison Bundles

The key point for this operand is to build the ===(EQ) , =/=(NEQ) check into circuit. Luckily, the design of Chisel makes it possible for us to do it on Scala level. Chisel saved all name -wire binds for a bundle in a ListMap[String, Data]. We extracted it from standard Chisel, and use the information in it to build our compare circuit.

The difficult part of this function is that Chisel Bundles can be member of Chisel Bundles, which is to say, a recursive Bundle is legal. For example:

```
class MySignedInt() extends Bundle{
  val sign = UInt(1.W)
  val num  = UInt((31).W)
}
class ComplexBundle() extends Bundle{
  val int = new MySignedInt()
}
```

Therefore, a recursive match is implemented to compare Bundles in Bundles.

4.3 Workflow of the Whole Implementation

The general workflow of the whole function looks like that:

```
Check the type of the inputs: Data type match{
    case bundle -> recursive EQ/NEQ check, build circuit and return
    case not bundle ->:
        Case Reg : warning/exception
        Case _ :
            Type match test
            Width match test
            build circuit and return
}
```

5. Evaluation

In the template project, we have designed built-in unit test for this package. It is consisted of three different parts: CounterTester, WidthMismatchTester and BundleMatchTester. A list of tests is shown below:

CounterTester will check stand chisel's behavior. pliChiselImprove is not imported.

1. standard chisel counter: Test the function of a standard chisel counter.

WidthMismatchTester will check if the width check for basic data type runs correctly.

1. UInt-UInt width dismatch counter
2. UInt-UInt width match counter
3. UInt-Reg width dismatch module in standard chisel
4. UInt-Reg width match module in standard chisel
5. UInt-Reg width dismatch module in pliChiselImprove
6. UInt-Reg width match module in in pliChiselImprove
7. SInt-SInt width dismatch module
8. SInt-SInt width match module

BundleMatchTester will check if the package can deal with bundle === and =/= correctly

1. bundle match circuit: EQ
2. bundle match circuit: NEQ
3. bundle match circuit: recursive EQ
4. bundle match circuit: recursive NEQ

You can run these tests in the template project by running the following commands in sbt:

```
testOnly pliChiselImprove.CounterTester
testOnly pliChiselImprove.BundleMatchTester
testOnly pliChiselImprove.WidthMismatchTester
```

To make this report more reader friendly we made it 6 pages.
Trust us we can compress it to 5 pages if we do not put different sections in different pages :)