Chisel: Introduction

Muhammad Tahir*

Lecture 1

*Professor, Electrical Engineering Department, University of Engineering and Technology Lahore

Contents

Scala Programming

- Purely object oriented, but with functional capabilities
- Seamless Java interoperability, classes and objects are similar to Java
- But "everything is an object"
- Type inference and powerful parameterization
- Open source (BSD License)

Scala keywords: val and var

- `val` is
  - *immutable*, once an object is assigned, it can not be replaced or reassigned
  - similar to *constant* keyword in C
  - immutable, however, the state of assigned object can change
- `var` is
  - *mutable*, an object assigned can be replaced through out its life
  - similar to any *variable* in C
  - mutable, but previously assigned object can be replaced with another object of same type (or should be type-casted)

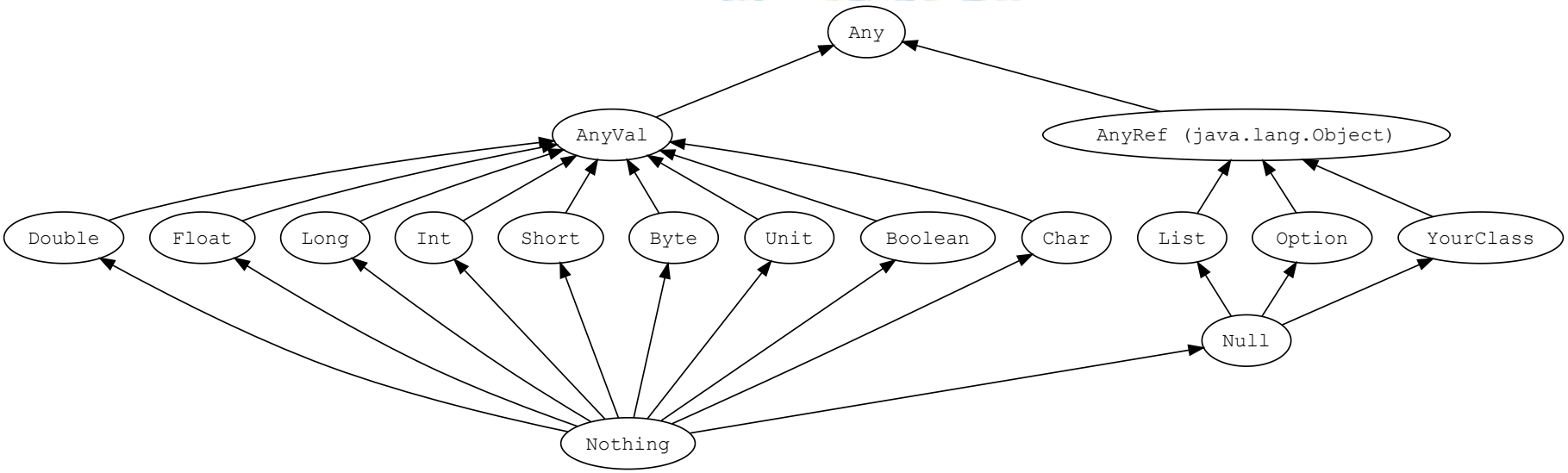
Scala Datatypes

Table: Data types in Scala.

Data type	Description
Byte	8-bit signed two's complement integer
Short	16-bit signed two's complement integer
Int	32-bit signed two's complement integer
Long	64-bit signed two's complement integer
BigInt	128-bit signed two's complement integer
Char	16-bit unsigned unicode charater
String	A sequence of chars
Float	32-bit single-precision float
Double	64-bit double-precision float
Boolean	true or false

Scala Datatypes Cont'd

Subset of Scala datatypes hierarchy*



* The figure is retrieved from:

https://docs.scala-lang.org/tour/unified-types.html

Scala Classes

```
class Counter(counterBits: Int) {
    val max = (1 << counterBits) - 1
    var count = 0

    if(count == max) {
        count = 0
    }
    else {
        count = count + 1
    }
    println(s"counter created with max value $max")
}
```

Listing 1: Scala Counter class.

Scala Type Cast

- Scala uses `asInstanceOf[ ]` for type cast
- Can be used to cast numeric data

```
val f: Float = 34.6F;
val c: Char = 'c';

val ccast = c.asInstanceOf[Int];
val fcast = f.asInstanceOf[Int];

display("Char ", c);
display("Char to Int ", ccast);

display("Float ", f);
display("Float to Int ", fcast);

def display[A](y: String, x: A): Unit = {
     println(
     y + " = " + x + " is of type " +
     x.getClass
     );
}
```

Scala Type Cast Cont'd

```
class Parent {
      val countP = 10
      def display(): Unit = {
            println("Parent counter : " + countP);
      }
}
class Child extends Parent {
      val countC = 12
      def displayC(): Unit = {
            println("Child counter : " + countC);
      }
}

object Top {
      def main(args: Array[String]): Unit =
      {
            var  pObject = new Parent()                          // parent object
            var  cObject = new Child()                           // child object
            var  castedObject = cObject.asInstanceOf[Parent] // object cast
            pObject.display()
            cObject.display()
            cObject.displayC()
            castedObject.display()
      }
}
```

Chisel

`Chisel`: Constructing Hardware in a Scala Embedded Language

- Chisel is simply a set of predefined special class definitions, objects and usage conventions within Scala
- A Chisel program is actually a Scala program
- (Chisel) embeds hardware construction primitives within an existing language (Scala)
- Compiled output constructs the hardware modules

Chisel Datatypes

- Datatypes specify the type of values held in state elements (*register* or *memory*) or flowing on *wires*
- There are three primitive datatypes
  - UInt — Unsigned integer
  - SInt — Signed integer (different from Scala `Int`)
  - Bool — Binary value (different from Scala `Boolean`)

Chisel Datatypes Cont'd

Constructing constants or defining data using primitive datatypes

```
Bool -- true.B or false.B

UInt -- 1234.U    // decimal value
UInt -- "b10010" // 5 bit binary literal from string
UInt -- "h3F".U  // 8 bit hexa literal from string

SInt -- -37.S     // -ve signed value
SInt -- 107.S     // +ve signed value
```

Literal definitions using primitive data types and type casting

```
// constant/literal definitions
val x1 = 23.S(32.W)       // x1 = 0x0000 0017

val y1 = (23.U).asSInt    // y1 = 23, width inferred
```

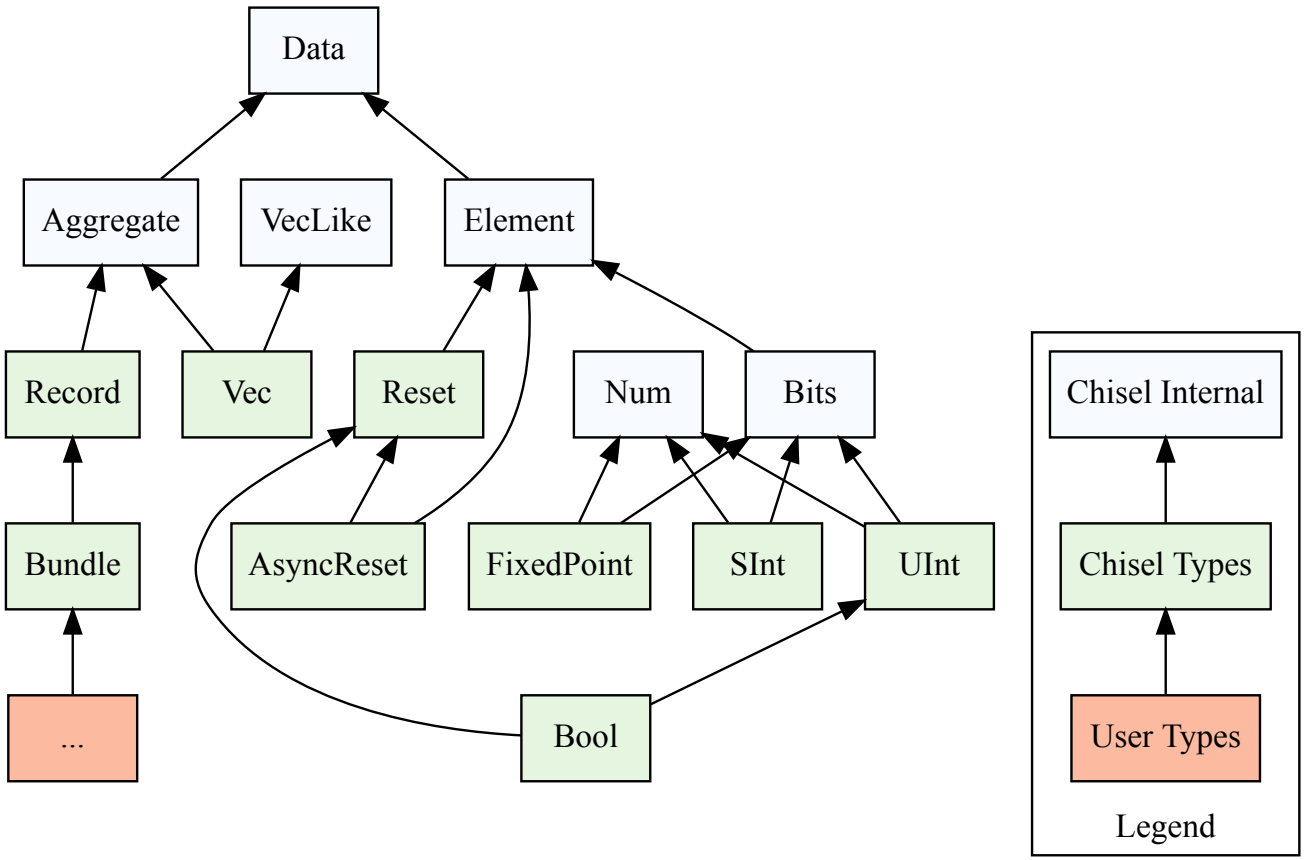
Chisel Datatypes Cont'd

Signal definitions

```
// data definitions
val s1 = WireInit(true.B) // Bool, initialized
val s2 = Wire(Bool())     // Bool, uninitialized

val x1 = WireInit(-45.S(8.W)) // SInt, initialized 8-bit
val x2 = WireInit(-45.S) // SInt, initialized width inferred
val x3 = Wire(SInt()) // SInt, uninitialized width inferred

val y1 = WireInit(102.U(8.W)) // UInt, initialized 8-bit
val y2 = WireInit(102.U) // UInt, initialized width inferred
val y3 = Wire(UInt()) // UInt, uninitialized width inferred

val z1 = Wire(Bits()) // Bits, uninitialized width inferred
val z2 = Wire(Bits(16.W)) // Bits, uninitialized 16-bit
```

Chisel Datatypes Cont'd

Chisel base datatypes* and their hierarchy



* The figure is retrieved from:

https://github.com/freechipsproject/chisel3

Scala-land vs Chisel-land

- **Chisel** code provides circuit related constructs and is compiled to construct the hardware modules (Verilog code)
- **Scala** is responsible for filling the values, book keeping and management of Chisel modules
- **Chisel** provides blackboxes to integrate third party IP or an optimized Verilog module
- Chisel and Scala have different data types
- Chisel is built on Scala and Scala is built on Java

Chisel Counter Example

Introduction to package, class and object, and hardware constructs (Chisel), by example

```
import chisel3._

class Counter(counterBits: UInt) extends Module {
    val max = (1.U << counterBits) - 1.U
    val count = RegInit(0.U(16.W))

    when(count === max) {
        count := 0.U
    }.otherwise{
        count := count + 1.U
    }
    println(s"counter created with max value $max")
}
```

Listing 2: Chisel based counter implementation.

Chisel Counter Example Cont'd

Filling the missing links in the previous example

```
import chisel3._

class Counter(counterBits: UInt) extends Module {
    val io = IO(new Bundle {
        val result = Output(Bool())
    })

    val max = (1.U << counterBits) - 1.U
    val count = RegInit(0.U(16.W))

    when(count === max) {
        count := 0.U
    }.otherwise{
        count := count + 1.U
    }
    io.result := count(15.U)
    println(s"counter created with max value $max")
}
```

Generating Hardware

Let us generate the Verilog code

```
import chisel3._
import chisel3.stage.ChiselStage

class AdderWithOffset extends Module {
    val io = IO(new Bundle {
        val x   = Input(SInt(16.W))
        val y   = Input(UInt(16.W))
        val z   = Output(UInt(16.W))
    })

    // Initialized as UInt, casted to SInt, type is inferred
    val y1 = (23.U).asSInt    // width will be optimized
    val in1 = io.x + y1
    io.z := in1.asUInt + io.y // Typecast SInt to UInt
}
println((new ChiselStage).emitVerilog(new AdderWithOffset))
```

Width Inference and Its Optimization

Notice the optimization applied at line 10 in the below code listing

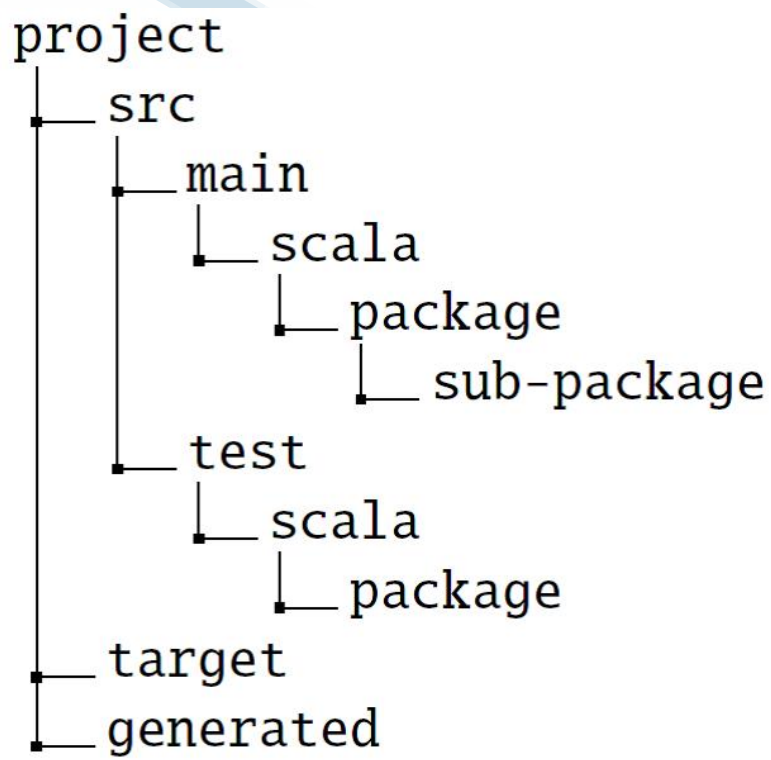
```
1  module AdderWithOffset(
2  input          clock,
3  input          reset,
4  input  [15:0]  io_x,
5  input  [15:0]  io_y,
6  output [15:0]  io_z
7  );
8
9  wire [15:0] _T_2; // @[main.scala 16:13]
10 assign _T_2 = $signed(io_x) - 16'sh9; // @[main.scala 16:13]
11 assign io_z = _T_2 + io_y; // @[main.scala 16:6]
12 endmodule
```

Counter Parameterization

```
import chisel3._
import chisel3.stage.ChiselStage

class Counter(size: Int, maxValue: UInt) extends Module {
    val io = IO(new Bundle {
        val result = Output(Bool())
    })
    // 'genCounter' with counter size 'n'
    def genCounter(n: Int, max: UInt) = {
        val count = RegInit(0.U(n.W))

        when(count === max) {
            count := 0.U
        }.otherwise {
            count := count + 1.U
        }
        count
    }
    val counter1 = genCounter(size, maxValue)
    io.result := counter1(size-1)
}
println((new ChiselStage).emitVerilog(new Counter(8,255.U)))
```

Project Organization

- Project hierarchy
- Device *generator* code in **main**
- Device *tester* code in **test**



Chisel Project and Errors

- Chisel Errors: Illegal in chisel that might be legal in Scala
- Scala Errors: Syntax, type mismatch etc., during compilation by Scala compiler
- FIRRTL errors: Errors found during transformations and generating Verilog
- Java Stack Trace: Underlying implementation encountered and exception and crashed

Reading List I

- Read Chapters 1 to 3 of [Odersky et al., 2016] for an introduction to Scala
- Read Chapter 2 of [Schoeberl, 2019] for basic know how of Chisel and Chapter 1 for tools installation

References


Odersky, M., Spoon, L., and Venners, B. (2016).
Programming in Scala.
Artima Incorporation.


Schoeberl, M. (2019).
Digital Design with Chisel.
Kindle Direct Publishing.