CHISEL Labs

LAB NO:	01	TOPIC:	Introduction to Scala and Chisel	↓

Compiled by: Shahzaib Kashif

Lab 01: Introduction to Scala and Chisel

Objective:

This laboratory session introduces Scala programming and the Chisel hardware description language (HDL) embedded in Scala. The goal is to enable you to write simple Scala programs and use Chisel to design basic hardware modules. This lab has been updated for compatibility with the latest Chisel version (7.0.0 as of August 2025), including minor syntax adjustments to avoid deprecations.

Note on Setup

To run Chisel code, use Scala Build Tool (sbt). Create a build.sbt file with below snippet or simply clone merledu/Scala-Chisel-Learning-Journey:

build.sbt

```
scalaVersion := "3.4.2"  // Latest Scala 3 for Chisel 7 compatibility
libraryDependencies ++= Seq(
  "org.chipsalliance" %% "chisel" % "7.0.0-RC3",
  "org.chipsalliance" %% "chisel-plugin" % "7.0.0-RC3" cross CrossVersion.full
)
```

Compile and run with `sbt run`

1.1 Introduction to Scala

Scala is a high-level language combining object-oriented and functional programming paradigms. Scala source code compiles to Java bytecode and runs on the Java Virtual Machine (JVM). It is interoperable with Java. We start with primitive data types, followed by object-oriented programming concepts like classes and objects.

1.1.1 Scala Data Types

n Scala, all values have an associated data type, including numbers and functions. Data types (Table 1.1) are objects. Objects can be immutable (val) or mutable (var). Use val for hardware in Chisel to ensure constancy after elaboration; var for tests or variables.

Table 1.1: Scala data types.

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	Arbitrary-precision signed integer
Char	16-bit unsigned Unicode character
String	A sequence of chars
Float	32-bit single-precision float
Double	64-bit double-precision float
Boolean	true or false

Data types follow a hierarchy (Figure 1.1). Any is the supertype, with subclasses AnyVal (value types) and AnyRef (reference types).

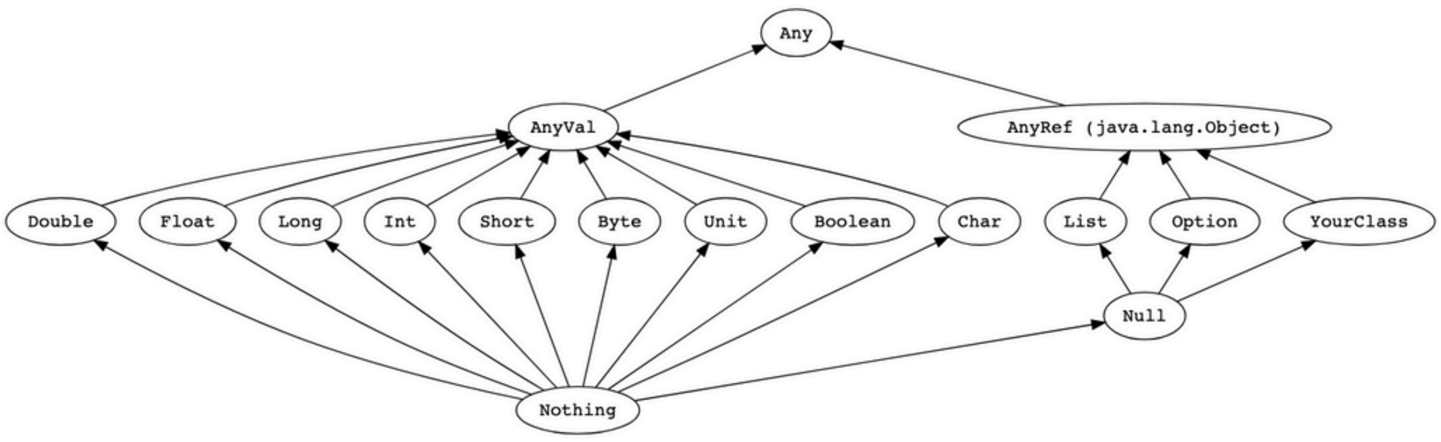


Figure 1.1: Subset of Scala data types hierarchy. (Source: https://docs.scala-lang.org/tour/unified-types.html)

1.1.2 Scala Classes and Objects

Classes are defined with class keyword. Example: A simple counter class (Codeblock 1.1). Objects are instances; singleton objects use object.

Counter.scala

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

  def increment(): Unit = {
    if (count == max) {
      count = 0
    } else {
      count = count + 1
    }
  }
  println(s"Counter created with max value $max")
}
```

Codeblock 1.1: Scala class description.

1.1.3 Scala Type Casting

Use `asInstanceOf[T]` for casting. Numeric example (Codeblock 1.2); object casting (Codeblock 1.3) allows upcasting (child to parent) but not downcasting without checks.

TypeCasting.scala

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

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

def display[A](label: String, value: A): Unit = {
  println(s"$label = $value is of type ${value.getClass}")
}

display("Char", c)
display("Char to Int", ccast)
display("Float", f)
display("Float to Int", fcast)
```

Codeblock 1.2: Scala numeric type cast.

TypeCasting.scala

```scala
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 = {
    val pObject = new Parent()
    val cObject = new Child()
    val castedObject = cObject.asInstanceOf[Parent]

    pObject.display()
    cObject.display()
    cObject.displayC()
    castedObject.display()
  }
}
```

Codeblock 1.3: Scala object type cast.

1.2 Introduction to Chisel

Chisel (Constructing Hardware in a Scala Embedded Language) is a DSL in Scala for hardware design. A Chisel program elaborates to FIRRTL, which compiles to Verilog. Updated for Chisel 7: Avoid deprecated .asTypeOf by using Wire(...) or Reg(...) directly.

1.2.1 Chisel Datatypes

Chisel types for hardware: `UInt` (unsigned), `SInt` (signed), Bool. Use literals like 23.U, -45.S. Widths inferred or specified (e.g., 8.W).

ChiselLiterals.scala

```scala
// Literals
val x1 = 23.S(32.W)   // Signed 32-bit
val y1 = 23.U.asSInt   // UInt to SInt
```

Codeblock 1.4: Defining literals in Chisel.

SignalDefinitions.scala

```scala
// Signals
val s1 = WireInit(Bool(), true.B)  // Initialized Bool
val s2 = Wire(Bool())              // Uninitialized

val x1 = WireInit(-45.S(8.W))      // Initialized SInt
val y1 = WireInit(102.U(8.W))      // Initialized UInt
val z1 = Wire(Bits(16.W))          // Uninitialized Bits
```

Codeblock 1.5: Signal definitions

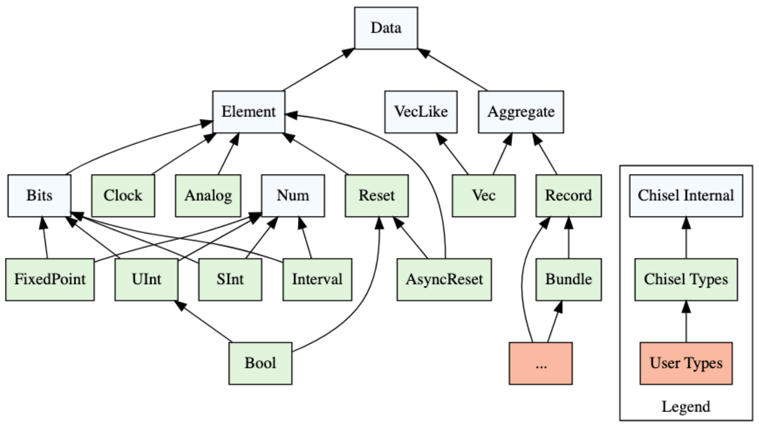


Figure 1.2: Chisel base data types hierarchy.

1.2.2 Counter Class Revisited

Chisel counter module (Codeblock 1.6 partial, 1.7 complete). Extends Module; mandatory io Bundle.

CounterRevisited.scala

```scala
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")
}
```

Codeblock 1.6: Chisel counter partial (elaboration-time print).

CounterRevisited.scala

```scala
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)
  println(s"Counter created with max value $max")
}
```

Codeblock 1.7: Chisel counter complete implementation.

To generate Verilog: Use `chisel3.stage.ChiselStage.emitVerilog(new Counter(8.U))`

1.3 Optimization of Signals and Parametrized Hardware Generation

Chisel optimizes during elaboration. Example: Constant folding (Codeblock 1.8).

Adder.scala

```scala
import chisel3._

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))
  })

  val y1 = 23.U.asSInt  // UInt to SInt
  val in1 = io.x + y1
  io.z := in1.asUInt + io.y
}

println(chisel3.stage.ChiselStage.emitVerilog(new AdderWithOffset))
```

Codeblock 1.8: Data optimization (generates Verilog with folded constants).

1.3.1 Parametrized Hardware Generation

Use functions for reusable logic (Codeblock 1.9).

Adder.scala

```scala
import chisel3._

class Counter(size: Int, maxValue: UInt) extends Module {
  val io = IO(new Bundle {
    val result = Output(Bool())
  })

  def genCounter(n: Int, max: UInt): 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(chisel3.stage.ChiselStage.emitVerilog(new Counter(8, 255.U)))
```

Codeblock 1.9: Parametrized Chisel counter.

1.4 Exercises

Exercise 1: Modify the counter in Codeblock 1.7 to use SInt for count (handle signed wrapping carefully).

Exercise 2: Reset the counter when its MSB changes from 0 to 1 (use a register to track previous MSB).

Exercise 3: Change max in Codeblock 1.9 to Int type, then cast to UInt (e.g., maxValue.asUInt).

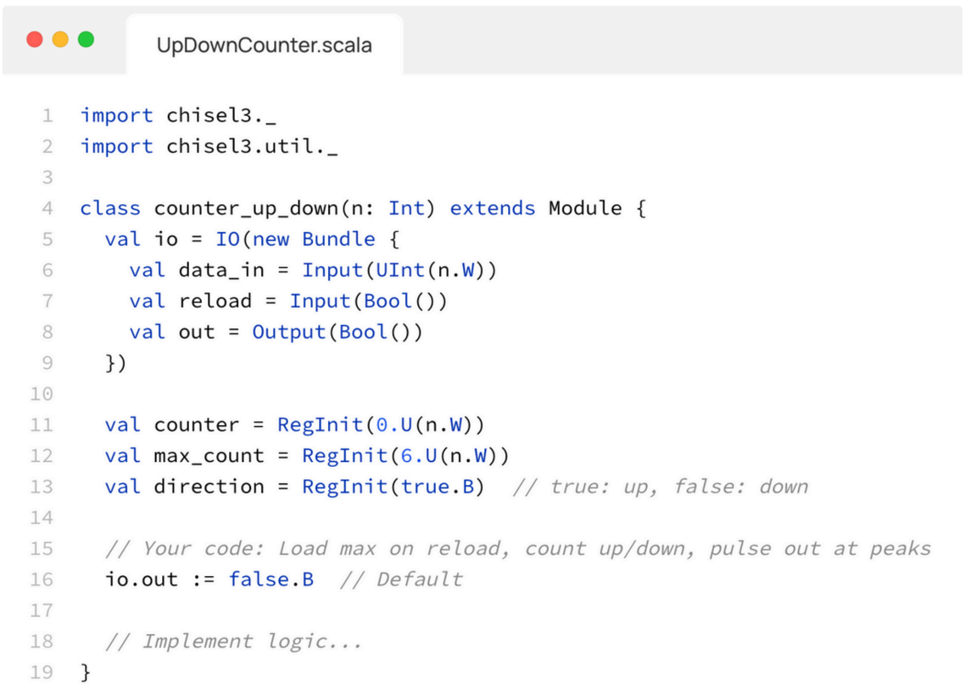
1.5 Assignments

Task 1: Determine feasible datatype castings (update table for Chisel 7 compatibility, e.g., Bool to UInt is possible via asUInt).

Table 1.2: Different groups of hardware operations.

1st Type	2nd Type	Possible?	If not, why?	Languages
SInt	SInt			
SInt	UInt			
UInt	UInt			
Clock	UInt			
UInt	SInt			
Bool	UInt			
Bool	Int			
UInt	Int			
SInt	Int			
Int	SInt			
Int	UInt			

Task 2: Implement an up-down counter. Counts up to data_in, then down to 0; sets out high for one cycle at max/min. Use reload to load data_in.


UpDownCounter.scala

```scala
import chisel3._
import chisel3.util._

class counter_up_down(n: Int) extends Module {
  val io = IO(new Bundle {
    val data_in = Input(UInt(n.W))
    val reload = Input(Bool())
    val out = Output(Bool())
  })

  val counter = RegInit(0.U(n.W))
  val max_count = RegInit(6.U(n.W))
  val direction = RegInit(true.B)  // true: up, false: down

  // Your code: Load max on reload, count up/down, pulse out at peaks
  io.out := false.B  // Default

  // Implement logic...
}
```

Codeblock 1.10: Skeleton code for counter implementation.