Experiment 1

Introduction to Scala and Chisel

Objective

This laboratory session will be an introduction to Scala programming and the Chisel library embedded in Scala. The objective is to enable the reader to write simple Scala programs and use Chisel library.

1.1 Introduction to Scala

Scala is a high-level language, which combines object-oriented and functional programming. Scala source code is compiled to Java bytecode and the generated executable runs on Java Virtual Machine (JVM). Scala is inter-operable with Java. We will start with primitive data types in Scala followed by an introduction to object oriented programming, discussing classes and objects.

1.1.1 Scala Data Types

In Scala, all values have an associated data type, which includes numerical values as well as functions. All data of different types, as listed in Table 1.1, are treated as *objects* in Scala [**?**]. Each data object can be immutable (`val` type) or mutable (`var` type). A value can be reassigned to a mutable object during elaboration but it cannot be done with an immutable object. Though new values cannot be reassigned to immutable objects once they are assigned, yet the state of assigned object can change. When constructing hardware modules using Chisel library, we will define objects (for data as well as for class instances) using `val`. However, to write unit tests, both `val` and `var` will be used.

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	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

Different data types listed in Table 1.1 follow a type hierarchy. A subset of this data type hierarchy is depicted in Figure 1.1[1]. The `Any` is supertype in Scala and has two direct subclasses, namely, `AnyVal` and `AnyRef`.

[1]The figure is retrieved from: https://docs.scala-lang.org/tour/unified-types.html

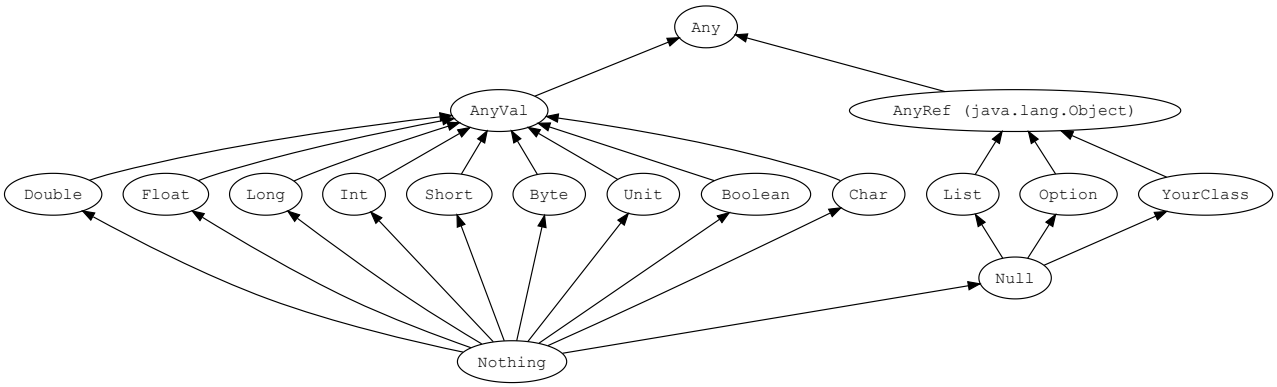

Figure 1.1: Subset of Scala data types hierarchy.

1.1.2 Scala Classes and Objects

Defining a `class` in Scala is illustrated in Listing 1.1. A class is declared using the keyword `class` followed by the name of the class, which is `Counter` in this case. Parameter `counterBits` is passed in round brackets. Variable `max` is declared with `val` so it cannot be reassigned. The rest of the logic is simple, it just adds one to `count` variable until the count reaches maximum, when it resets to zero.

An instance of a class is called an `object`. The keyword object is used to describe singleton objects, which has only one instance. It can also be considered as defining a class and instantiating it only once.

```
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 1.1: Scala class description.

1.1.3 Scala Type Casting

Scala uses `asInstanceOf[]` method for type casting of numeric data as well as `object` casting. Listing 1.2 illustrates numeric data typecasting. We can also perform casting of objects. An object of a child (extended or derived) class can be casted to that of a parent class but not the other way around. This is illustrated in Listing 1.3.

```
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
    );
}
```

Listing 1.2: Scala numeric type cast.

```
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()
    }
}
```

Listing 1.3: Scala object type cast.

1.2 Introduction to Chisel

Chisel (Constructing Hardware In a Scala Embedded Language) is simply a set of predefined special class definitions, objects as well as usage conventions within Scala. A Chisel program [**?**] is actually a Scala program, which constructs the hardware modules when compiled.

1.2.1 Chisel Datatypes

In Chisel, datatypes specify the type of values held in state elements (register or memory) or flowing on wires. The datatypes in Chisel are different from the ones in Scala. In some cases, we may need to cast (typecast) between Scala and Chisel types. Furthermore, casting between Chisel types may also be required.

Unsigned and signed integers are represented by the keywords UInt and SInt respectively. Boolean values are defined using Bool. Listings 1.4 and 1.5 provide some illustrations for Chisel data types.

```
// constant/literal definitions

val x1 = 23.S(32.W)      // x1 = 0x0000 0017
//.W with a constant value is used to define with of x1. If round backets
   are left empty then width will be inferred

val y1 = (23.U).asSInt  // y1 = 23, width inferred //.asSInt is used to
   convert into signed integer.
```

Listing 1.4: Defining literals/constants in Chisel.

```
// signal 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
```

Listing 1.5: Signal definitions of different datatypes.

Figure 1.2[2] shows the base data types and their hierarchy in Chisel that can be used to define different circuit components.

1.2.2 Counter Class Revisited

Below we implement the Counter class again using Chisel library to generate the corresponding hardware module.

```
import chisel3._
```

[2]The figure is retrieved from: https://github.com/freechipsproject/chisel3

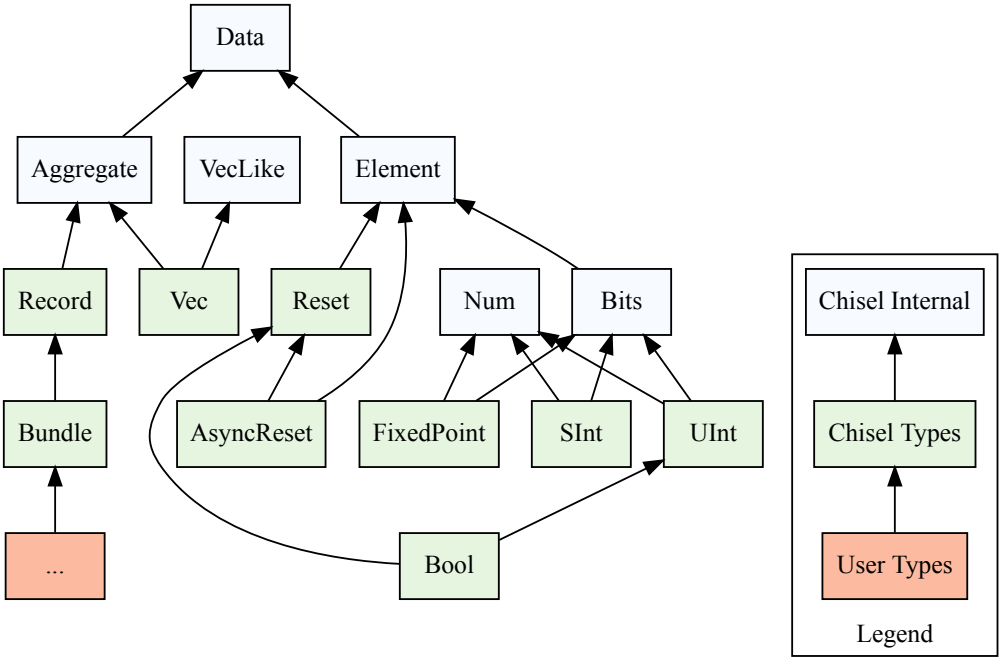

Figure 1.2: Chisel base data types and their hierarchy.

```
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 1.6: Chisel counter partial implementation.

However, the above counter implementation will not compile due to some mandatory Chisel syntax requirements being missed out. One such requirement is an IO method that can only be omitted in an *abstract* class. The next implementation eliminates this limitation and can be used to generate the counter module.

```
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")
}
```

Listing 1.7: Chisel counter complete implementation.

1.3 Optimization of Signals and Parametrized Hardware Generation

In the Listing 1.8, variable 'y1' is first initialized with unsigned integer 23 then it is converted to signed number 9 which is 2's complement of 23. In the FIRRTL[3] generated verilog 9 will be subtracted from the input io.x.

```
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))
    })

    // Initialized as UInt and casted to SInt
    val y1 = (23.U).asSInt
    val in1 = io.x + y1
    io.z := in1.asUInt + io.y  // Typecast SInt to UInt
}

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

// The generated Verilog code
module AdderWithOffset(
input          clock,
input          reset,
input  [15:0] io_x,
input  [15:0] io_y,
output [15:0] io_z
);
wire [15:0] _T_2;
assign _T_2 = $signed(io_x) - 16'sh9;
assign io_z = _T_2 + io_y;
endmodule
```

[3]FIRRTL will be discussed in Experiment 4.

Listing 1.8: Data optimization

1.3.1 Parametrized Hardware Generation

Functions are defined using keyword `def`. Similar to the class declaration, it also has a name followed by parameters list. Every function has a return type. If the return type is not defined during declaration then the last line of the function block will be the returned value and its type will be inferred.

Based on given parameters, Chisel code will be configured accordingly. For instance, in Listing 1.9, 'size' and 'maxValue' parameters will configure the counter hardware for bitwidth and reload value, respectively. For size = 8 and maxValue = 255 a counter register with width '8' will be initialized with zero value and parameter maxValue will set the maximum value at which counter will restart the count. One bit output is derived from the most significant bit (MSB) of the counter register, which can be used as a clock divisor.

```
import chisel3._

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
    }

    // genCounter instantiation
    val counter1 = genCounter(size, maxValue)
    io.result := counter1(size-1)
}

println((new chisel3.stage.ChiselStage).emitVerilog(new Counter(8, 255.U)))
```

Listing 1.9: Chisel counter implementation.

1.4 Exercises

Exercise 1: Modify the counter in Listing 1.7 to use SInt type count.

Exercise 2: Make the counter to reset its count to 0 when its MSB (most significant bit) changes from 0 to 1.

Exercise 3: Modify the counter in Listing 1.9 to make max parameter of type Int and then use typecasting to make it work.

1.5 Assignments

Task 1: Find out which of the following datatype castings are possible.

Table 1.2: Different groups of hardware operations.

1st type	2nd type	Possible or not	If not, then why	1st type language - 2nd type language
SInt	SInt			
SInt	UInt			
UInt	UInt			
Clock	UInt			
UInt	SInt			
Bool	UInt			
Bool	Int			
UInt	Int			
SInt	Int			
Int	SInt			
Int	UInt			

Task 2: Define a class in Scala that implements an up-down counter. The counter starts from 0, counts up to a pre-defined value and then counts down to zero. It must repeats it counting and set io.out to high for one clock cycle when it reach either maximum or minimum values.

```
package Counter

import chisel3._
import chisel3.util._
import java.io.File

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))

    //Your code

}
```

Listing 1.10: Skeleton code for counter implementation.