Chisel Programming

Laboratory Manual

prepared by

Muhammad Tahir[1]
Waleed Bin Ehsan
Junaid Ahmed
Umair Riaz
Farhan Aslam
Owais Farooq
Hassaan Khalid

[1]Professor, Electrical Engineering Department, University of Engineering and Technology Lahore

Contents

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 [3]. 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

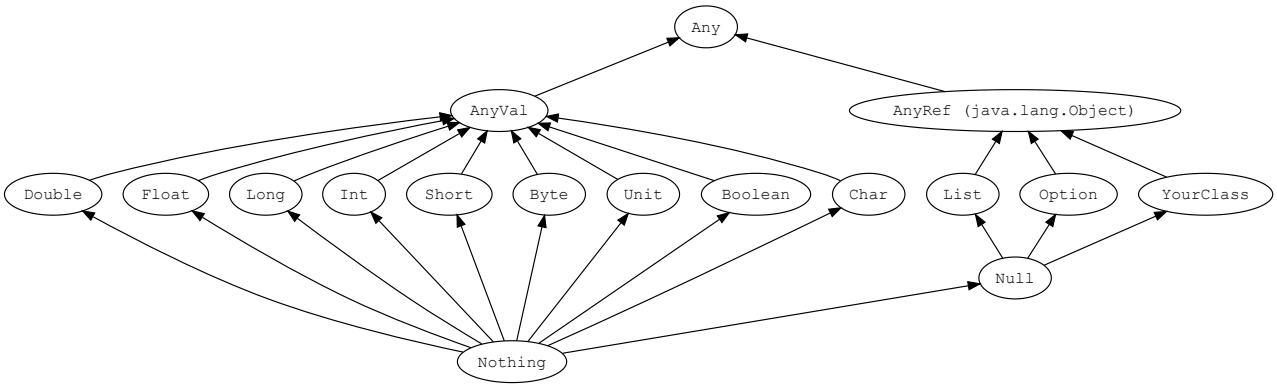


Figure 1.1: Subset of Scala data types hierarchy.

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 [4] 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

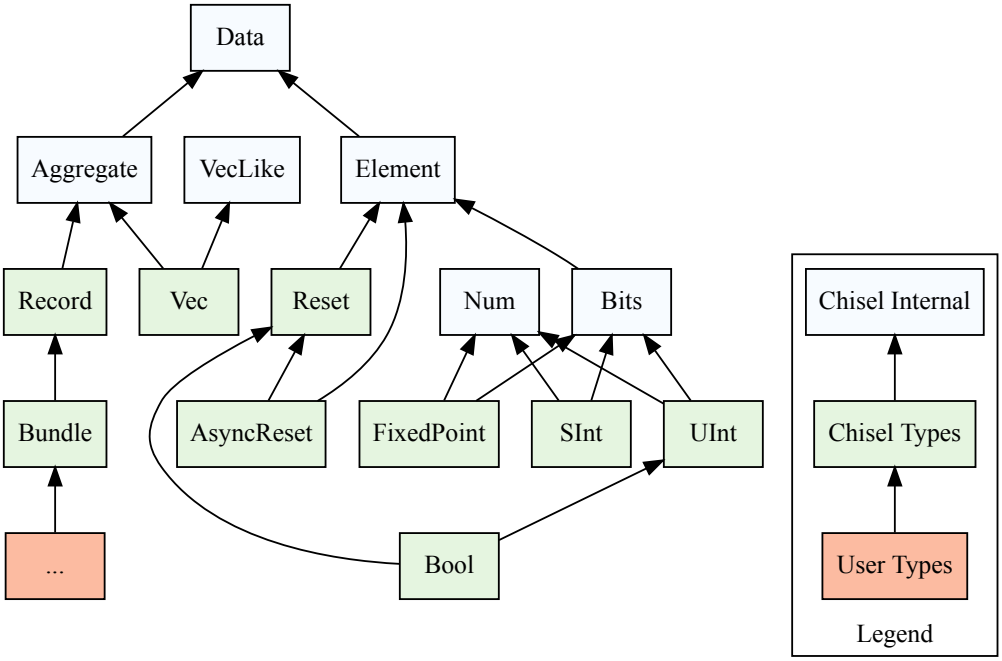


Figure 1.2: Chisel base data types and their hierarchy.

```
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 data type-casts 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 repeat its 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.

Experiment 2

Combinational Circuits

Objective

Familiarize with basic building blocks to design and implement combinational hardware, specifically with Mux and its variants.

2.1 Chisel: Hardware Operations

First, let us take a look at a list of operators that can be used to modify any piece of hardware. Table 2.1 categorizes such operators in different groups and also mentions the type of operands that can be operated on.

Table 2.1: Different groups of hardware operators.

Operator Symbol	Description	Operand Type
&& || !	AND, OR, NOT (logical)	Bool
& | ~ ^	AND, OR, NOT, XOR (bitwise)	UInt, SInt, Bool
<< >>	shift left, shift right (sign extend for SInt)	UInt, SInt
+ −	addition, subtraction	UInt, SInt
* / %	multiplication, division, modulus	UInt, SInt
=== =/=	equal, not equal (returns Bool)	UInt, SInt
> >= < <=	different comparisons (returns Bool)	UInt, SInt

2.1.1 Width Inference

Furthermore, if the output width of a module is not specified, it can be inferred by the tools, while generating the hardware (verilog output). Width inference is a useful feature and does offer certain level of optimization with it. Table 2.2[1] lists the width inference for some of the hardware operations.

Table 2.2: Bit width inference.

Operation	Bit Width
out = in1 + in2	W(out) = max{W(in1), W(in2) }
out = in1 +& in2	W(out) = max{W(in1), W(in2) } + 1
out = in1 & in2	W(out) = max{W(in1), W(in2) }
out = in1 * in2	W(out) = W(in1) + W(in2)
out = in1 ≪ shift	W(out) = W(in1) + max(shift)
out = in1 ≫ shift	W(out) = W(in1) - min(shift)
out = Cat(in1 , in2)	W(out) = W(in1) + W(in2)

A few examples of arithmetic operations with only addition and subtraction are shown in Listing 2.1.

[1] W(x) represents the bit width of signal or wire x.

Note the use of suffix symbols (% and &) on the operators in order to manipulate specific hardware implementation attributes.

```
// Arithmetic operations

// Addition without width expansion
val sum = x + y // OR
val sum = x +%y

// Addition with width expansion
val sum = x +& y

// Subtraction without width expansion
val sum = x - y // OR
val sum = x -%y

// Subtraction with width expansion
val sum = x -& y
```

Listing 2.1: Arithmetic operations.

2.1.2 Bitfield Manipulation

Listing 2.2 illustrates implementations for bitfield manipulations. Chisel library functions or constructs are utilized to implement different bitfield operations. For instance, `Cat` is used to concatenate multiple bitfields, with its first operand taking the leftmost place in the resulting output. Similarly, AND reduction of bits can be performed using `bits.andR` as illustrated in Listing 2.2.

```
// Bitfield manipulations
val xMSB = x(31)        // when x is 32-bit
val yLowByte = y(7, 0)  // y is atleat 8-bit

// concatenates bitfields with first operand on left
val address = Cat(highByte, lowByte)

// replicate a string multiple times
val duplicate = Fill(2, "b1010".U) // "b10101010".U

// Bitfield reductions
val data = "b00111010".U
val allOnes = data.andR             // performs AND reduction
val anyOne = data.orR              // performs OR reduction
val parityCheck = data.xorR        // performs XOR reduction
```

Listing 2.2: Bitfield manipulations

Another useful Chisel utility is `BitPat` (bit pattern), which provides literals with masks. `BitPat` is used for generating bit patterns involving don't care bits. An equality comparison of a bit pattern generated using `BitPat` will ignore don't care bits as illustrated in Listing 2.3.

```
// BitPat example
// define partial opcodes for RISC V instructions
def opCode_BEQ = BitPat("b000?????1100011")
def opCode_BLT = BitPat("b100?????1100011")

// opcode matching with don't care bits
when(opCode_BEQ === "b000110001100011".U){
   // above comparison evaluates to true.B
   // user code
}
```

Listing 2.3: BitPat illustration.

2.2 Mux: A Simple Combinational Block

The Mux is an essential combinational block and the Chisel library provides quiet a few constructs for this. Listing 2.4 shows the implementation of one bit 2-to-1 multiplexer using binary operations. On the other hand Listing 2.5 implements the same 2-to-1 multiplexer with 32-bit wide inputs using the Mux construct from the Chisel library.

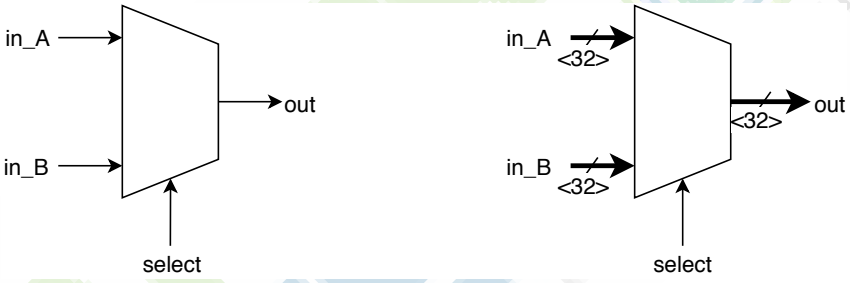


Figure 2.1: A simple 2 to 1 Mux.

```
import chisel3._

// Mux IO interface class
class Mux_2to1_IO extends Bundle {
   val in_A   = Input(Bool())
   val in_B   = Input(Bool())
   val select = Input(Bool())
   val out    = Output(Bool())
}
// 2 to 1 Mux implementation
class Mux_2to1 extends Module {
   val io = IO(new Mux_2to1_IO)

   // update the output
   io.out := io.in_A & io.select | io.in_B & (~io.select)
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_2to1()))
```

Listing 2.4: Mux 2-to-1 with scalar inputs.

```
import chisel3._

// Mux IO interface class
class Mux_2to1_IO extends Bundle {
    val in_A   = Input(UInt(32.W))
    val in_B   = Input(UInt(32.W))
    val select = Input(Bool())
    val out    = Output(UInt())
}
// 2 to 1 Mux implementation
class Mux_2to1 extends Module {
    val io = IO(new Mux2to1_IO)

    // update the output
    io.out := Mux(io.select, io.in_A, io.in_B)
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_2to1()))
```

Listing 2.5: Mux 2-to-1 with vector inputs.

2.2.1 Mux Tree

This is not a new construct, but illustrates different possible ways to wire multiple 2-to-1 multiplexers, to construct higher order multiplexers. You may think of it as nested multiplexers where the output of one multiplexer is the input of another multiplexer. Figure 2.2 shows two different implementations of a 4-to-1 multiplexer. It is important to notice that there are two key differences between these two implementations. First difference is in the number of selection lines. The second difference, which is more important, is the inherent priority of in_4 over in_3, in_2 and in_1 for the 4-to-1 multiplexer in Figure 2.2(b). Similarly for this very same multiplexer, the in_3 has priority over in_2 and in_1. In contrast, all the inputs to the multiplexer in Figure 2.2(a) are of equal priority.

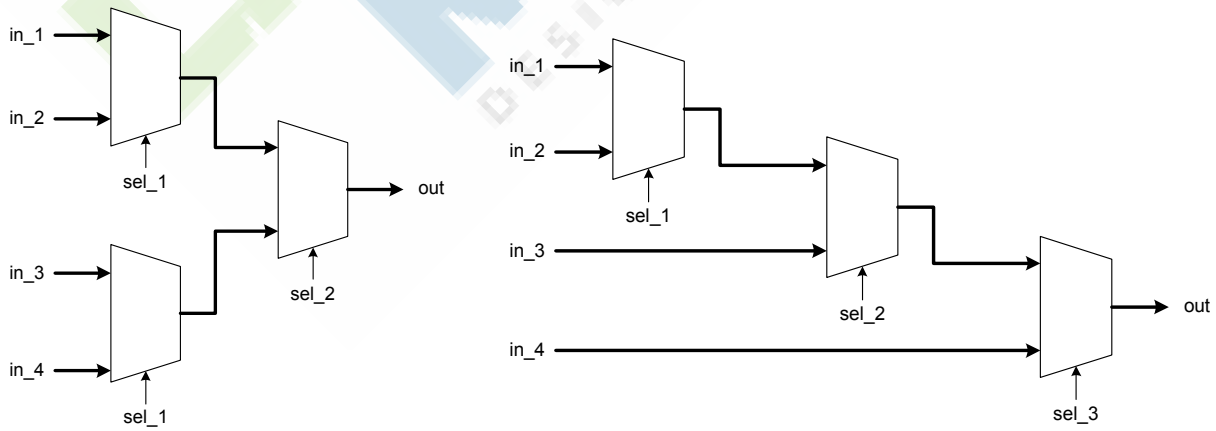


Figure 2.2: Two different 4-to-1 Mux implementations. (a) Equal priority inputs, (b) inputs with priority.

Listing 2.6 shows how an 8-to-1 mux, with equal priority inputs can be implemented. A 4-to-1 multiplexer with input priority is implemented in Listing 2.7. A multiplexer construct with input

priority, named PriorityMux, is available in the Chisel library as well.

```
// An 8-to-1 Mux example
import chisel3._

class LM_IO_Interface extends Bundle {
    val in  = Input(UInt(8.W))
    val s0  = Input(Bool())
    val s1  = Input(Bool())
    val s2  = Input(Bool())
    val out = Output(Bool())          // UInt(1.W))
}

class Mux_8to1 extends Module {
    val io = IO(new LM_IO_Interface)

    val Mux4_to_1_a = Mux(io.s1, Mux(io.s0, io.in(3), io.in(2)),
    Mux(io.s0, io.in(1), io.in(0)))
    val Mux4_to_1_b = Mux(io.s1, Mux(io.s0, io.in(7), io.in(6)),
    Mux(io.s0, io.in(5), io.in(4)))

    val Mux2_to_1 = Mux(io.s2, Mux4_to_1_b, Mux4_to_1_a)

    // Connecting output of 2_to_1 Mux with the output port.
    io.out := Mux2_to_1
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_8to1()))
```

Listing 2.6: Mux 8-to-1 with equal priority inputs.

```
// Mux with input priority
import chisel3._

class IO_Interface extends Bundle {
    val in  = Input(UInt(4.W))
    val s1  = Input(Bool())
    val s2  = Input(Bool())
    val s3  = Input(Bool())
    val out = Output(Bool())    // UInt(1.W))
}

class Mux_Tree extends Module {
    val io = IO(new IO_Interface)

    io.out := Mux(io.s3, io.in(3), Mux(io.s2, io.in(2),
    Mux(io.s1, io.in(1), io.in(0))))
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_Tree()))
```

Listing 2.7: Mux 4-to-1 with input priority.

2.2.2 MuxCase

If there are more than two input lines, we can use `MuxCase` as an alternate of instantiating multiple 2-to-1 multiplexers. This construct implements the same functionality as that of a multiplexer with equal priority inputs and each selection dependency is represented as a tuple in a Scala array. The basic syntax is shown below.

```
MuxCase(default, Array(c1 -> a, c2 -> b, ...))
```

Listing 2.8 implements an 8-to-1 Mux using `MuxCase`. The conditions can be written using Scala collection `Array` or `Seq`. We will discuss more about Scala collections in Exp 9.

Another important thing to note here is importing of another library, `chisel3.util._`, to use `MuxCase` or other variants of multiplexers available in the utilities package of Chisel library. We need to import this library in order to use these constructs.

```
// 8 to 1 mux using MuxCase
import chisel3._
import chisel3.util._

class MuxCase_ex extends Module {
    val io = IO(new Bundle{
        val in0 = Input(Bool())
        val in1 = Input(Bool())
        val in2 = Input(Bool())
        val in3 = Input(Bool())
        val in4 = Input(Bool())
        val in5 = Input(Bool())
        val in6 = Input(Bool())
        val in7 = Input(Bool())
        val sel = Input(UInt(3.W))
        val out = Output(Bool())
    })

    io.out := MuxCase(false.B, Array(
    (io.sel===0.U) ->     io.in0,
    (io.sel===1.U) ->     io.in1,
    (io.sel===2.U) ->     io.in2,
    (io.sel===3.U) ->     io.in3,
    (io.sel===4.U) ->     io.in4,
    (io.sel===5.U) ->     io.in5,
    (io.sel===6.U) ->     io.in6,
    (io.sel===7.U) ->     io.in7)
    )
}

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

Listing 2.8: MuxCase illustration for eight inputs.

2.2.3 MuxLookup

We observed that the input selection in `MuxCase` required boolean expressions to test the select signal. Use of these expressions can be avoided with the `MuxLookup` construct. Similar to `MuxCase`, its outputs and conditions are given as a Scala collection. The basic syntax is shown below.

```
MuxLookup(io.select, default, Array(c1 -> a, c2 -> b, ...))
```

Listing 2.9 implements an 8-to-1 Mux using `MuxLookup`.

```
// 8 to 1 mux using MuxLookup
import chisel3._
import chisel3.util._

class MuxLookup extends Module {
    val io = IO(new Bundle{
        val in0 = Input(Bool())
        val in1 = Input(Bool())
        val in2 = Input(Bool())
        val in3 = Input(Bool())
        val in4 = Input(Bool())
        val in5 = Input(Bool())
        val in6 = Input(Bool())
        val in7 = Input(Bool())
        val sel = Input(UInt(3.W))
        val out = Output(Bool())
    })

    io.out := MuxLookup(io.sel , false.B, Array(
    (0.U) ->    io.in0,
    (1.U) ->    io.in1,
    (2.U) ->    io.in2,
    (3.U) ->    io.in3,
    (4.U) ->    io.in4,
    (5.U) ->    io.in5,
    (6.U) ->    io.in6,
    (7.U) ->    io.in7)
    )
}

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

Listing 2.9: An 8-to-1 mux implementation using MuxLookup.

2.2.4 Mux1H

Another interesting multiplexer construct is the `Mux1H` (termed as Mux 1 hot). In Mux1H, the number of select lines is same as the number of inputs as shown in Figure 2.3. At any particular time instance only one select line should be high and each select line corresponds to one input. If more than one

select lines are high, the output is undetermined. Listing 2.10, shows the usage of Mux1H. For the `sel` signal values of 1, 2, 4 or 8, the corresponding output will be in0, in1, in2 or in3 respectively.

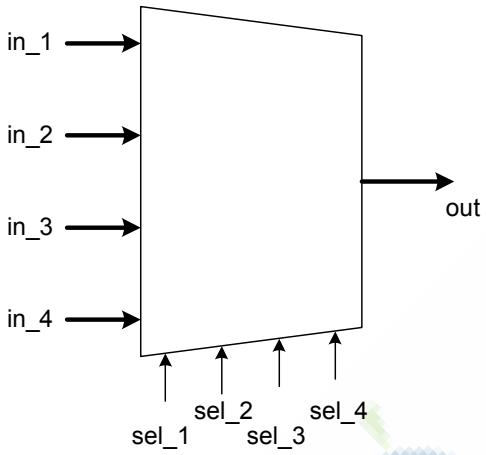


Figure 2.3: Block diagram for Mux1H.

```
// Mux-Onehot example
import chisel3._
import chisel3.util._

class mux_onehot_4to1 extends Module {
    val io = IO(new Bundle {
        val in0 = Input(UInt(32.W))
        val in1 = Input(UInt(32.W))
        val in2 = Input(UInt(32.W))
        val in3 = Input(UInt(32.W))
        val sel = Input(UInt(4.W))
        val out = Output(UInt(32.W))
    })

    io.out := Mux1H(io.sel, Seq(io.in0, io.in1, io.in2, io.in3))
}

println((new chisel3.stage.ChiselStage).emitVerilog(new mux_onehot_4to1()))
```

Listing 2.10: Mux1H illustration for four inputs.

2.3 Bundle and Vectors

If we have multiple input-output (IO) signals that need to be used by a module, we can conveniently put them together using `Bundle`. IO bundles can be constructed using a class that extends from `Bundle` with the signal directions defined explicitly.

Vectors are constructed using `Vec` and can be used to make an array of hardware or signals of same type. For instance, while defining an IO class, we can make an input to be a vector of length n. This will make the input to be an array of n elements. Listing 2.11 illustrates the use of bundles and vectors.

```
import chisel3._

class LM_IO_Interface extends Bundle{
    // Make an input from a Vector of 4 values
    val data_in = Input(Vec(4,(UInt(32.W))))

    // Signal to control which vector is selected
    val data_selector = Input(UInt(2.W))

    val data_out = Output(UInt(32.W))
    val addr = Input(UInt(5.W))

    // The signal is high for write
    val wr_en = Input(Bool())
}

class Mem_bundle_intf extends Module {
    val io = IO(new LM_IO_Interface)

    io.data_out := 0.U

    // Make a memory of 32X32
    val memory = Mem(32, UInt(32.W))

    when(io.wr_en){
        // Write for wr_en = 1
        // Write at memory location addr, with selected data from data_in (
         Vector)
        memory.write(io.addr, io.data_in(io.data_selector))
    } .otherwise{
        // Asyncronous read from addr location
        io.data_out := memory.read(io.addr)
    }
}

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

Listing 2.11: An example illustrating the use of Bundle and Vec.

2.4 Flipped and Bulk constructs

If we want to change the direction of any signal or a bundle of signals, we simply flip the bundle by applying the `Flipped` construct and the direction of all the signals is reversed i.e. from inputs are changed to outputs and vice versa. This is quite useful in scenarios where we are using bundles that are input to one module and output to another module.

We can also connect IO signals with same names across different modules by using the bulk connector `<>`. One scenario, where this would be useful, is a master/slave configuration as shown in Figure 2.4.

In master, we have a bundle which is input from a slave and the slave has the same connections as output to the master. We can connect the same bundle to the master and to the slave using the bulk connector. Listing 2.12 provides an illustration of using flipped and bulk connection.

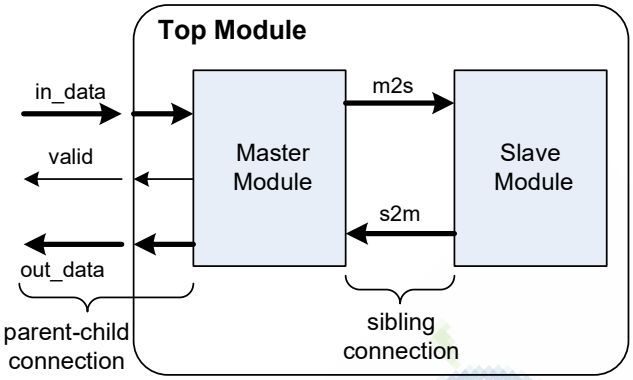


Figure 2.4: Bulk Connection Diagram

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

class Interface extends Bundle {
    val in_data  = Input(UInt(6.W))
    val valid    = Output(Bool())
    val out_data = Output(UInt(6.W))
}

class MS_interface extends Bundle {
    val s2m  = Input(UInt(6.W))
    val m2s  = Output(UInt(6.W))
}

class Top_module extends Module {
    val io = IO(new Interface)

    val master = Module(new Master)
    val slave  = Module(new Slave)
    //connecting top with master => same direction, same name connects
    io <> master.io.top_int
    //connecting master with slave => opposite direction, same name connects
    master.io.MS <> slave.io
}

class Master extends Module {
    val io = IO(new Bundle {
        val top_int = new Interface
        val MS = new MS_interface
    })
```

```
    io.MS.m2s := io.top_int.in_data
    io.top_int.valid := true.B
    io.top_int.out_data := io.MS.s2m
}

class Slave extends Module {
    val io = IO(Flipped(new MS_interface))

    io.s2m := io.m2s + 16.U
}

println(chisel3.Driver.emitVerilog(new Top_module))
```

Listing 2.12: Flipped and bulk connection example.

2.5 Exercises

Exercise 1: For vector Mux in Listing 2.5, devise a method that uses combinational circuit instead of the pre-defined Mux module (as has been done in Listing 2.4 for scalar Mux). The IO bundle must remain the same. (*Hint*: See Listing 2.2 for how to manipulate bits.)

Exercise 2: In Listing 2.6, an 8-to-1 Mux is created using Mux tree or nested Muxes while Listing 2.9 does the same but with MuxLookup. Try to alter Listing 2.9 by using nested MuxLookups. The first MuxLookup will contain only two branches each of which will contain another MuxLookup. These next level MuxLookups will contain four branches each.

Exercise 3: Refer to Listing 2.10 to create a 4-to-2 Encoder using Mux1H. (*Hint*: You may have to use pre-determined inputs instead of ports.)

2.6 Assignments

Task 1: Write Chisel code for a 5-to-1 multiplexer with specifications given in Table 2.3. A skeleton code is also given in Listing 2.13. Use it as a starting point. *Warning*: Write your code in the space specified for this purpose.

Table 2.3: 5-to-1 mux specifications

{s2,s1,s0}	Output
000	0.U
001	8.U
010	16.U
011	24.U
1xx	32.U

```
package Lab2

import chisel3._

class LM_IO_Interface extends Bundle {
```

```
    val s0  = Input(Bool())
    val s1  = Input(Bool())
    val s2  = Input(Bool())
    val out = Output(UInt(32.W))
}

class Mux_5to1 extends Module {
    val io = IO(new LM_IO_Interface)

    // Start coding here


    // End your code here
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_5to1))
```

Listing 2.13: 5-to-1 Mux skeleton code

Task 2: Write Chisel code for 4-bit Barrel shifter shown in Figure 2.5.

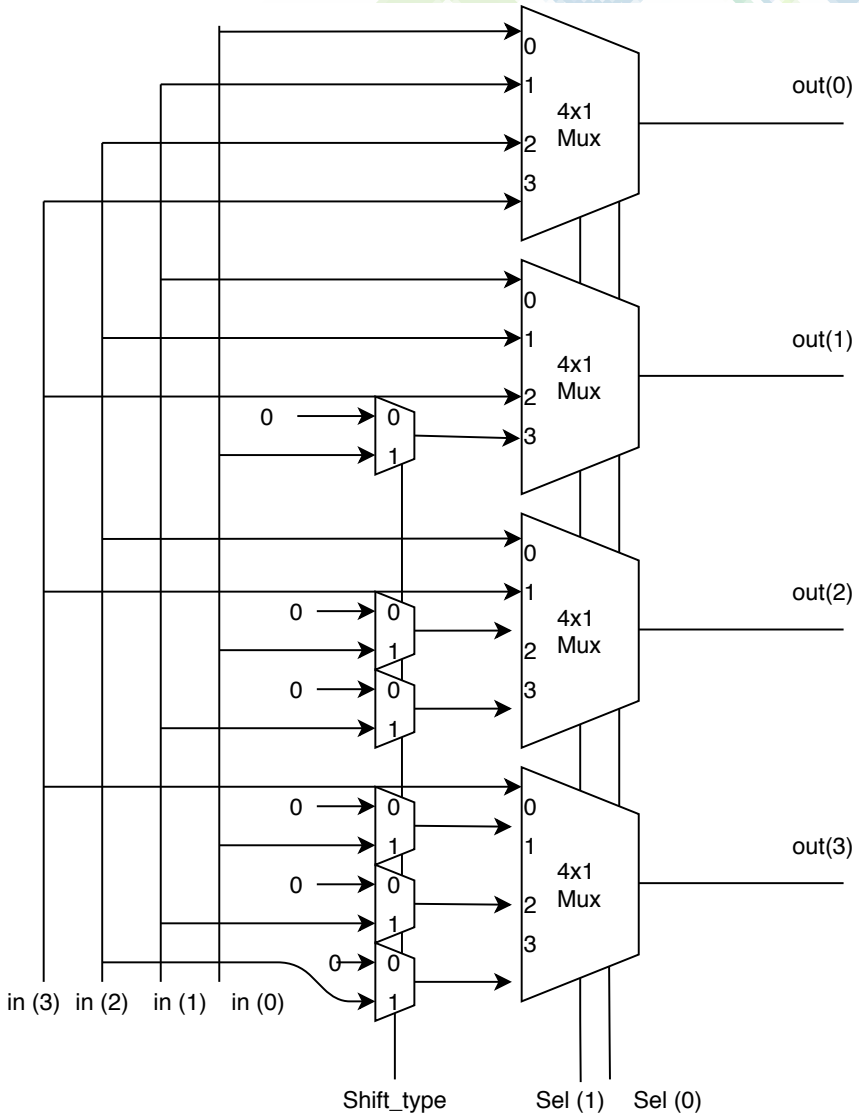


Figure 2.5: Four bit Barrel shifter.

```
package Lab2

import chisel3._
import chisel3.util._

class barrel_shift extends Module{
    val io = IO(new Bundle{
        val in = Vec(4, Input(Bool()))
        val sel = Vec(2, Input(Bool()))
        val shift_type = Input(Bool())
        val out = Vec(4, Output(Bool()))
    })

    // Start you code here


    // End your code here
}
println((new chisel3.stage.ChiselStage).emitVerilog(new barrel_shift))
```

Listing 2.14: 4 bit Barrel shifter skeleton code

Task 3: What are the hardware differences between `MuxCase` and `MuxLookup`.

Experiment 3

Control Flow and Combinational Circuits

Objective

To be able to perform] analysis and design of complex combinational circuits while using different control flow constructs.

3.1 Control Flow

The control flow is the order in which individual modules are enabled or selected to perform their respective operations. Control flow blocks allow us to enable a hardware block conditionally. Hardware control flow can be realized through different conditional constructs, which perform similar job. Chisel provides two conditional constructs, namely `when` block and `switch` block. Nesting of these conditional blocks is possible.

3.1.1 When, Elsewhen, Otherwise Construct

Similar to `if`, `else` in verilog, we can control the data flow by using `when`, `.elsewhen`, `.otherwise`. In Chisel, the `if`/`else` block is treated as a Scala conditional construct, which includes or excludes a hardware block in the generated hardware (i.e. in the emitted verilog). On the other hand, the working of `when` construct in Chisel is similar to `if`/`else` in verilog. The argument to `when` is a conditional expression that returns a `Bool`. An incompletely specified combinational output results in an error. One such possible scenario is when an unconditional update is not provided for a combinational output.

The `when` block is translated to Mux circuit, where the selection lines for Mux are dependent on the conditions used by `when` or `.elsewhen`.

```
when(/* condition for when */){
    /* do this if the condition for when is true */
}
.elsewhen(/* condition for elsewhen */){
    /* do this if the when is false and
    condition for .elsewhen is true. */
}
.otherwise{
    /* do this as a default condition, .otherwise
    do not require a condition as it is the default
    when no previous condition is met. */
```

```
}
```

The advantage of using .otherwise block is that we can omit the initialization, which other wise is required to avoid initialization error. For example, if we want to implement a decoder as shown in Figure 3.1 using when conditional construct, a possible implementation is illustrated in Listing 3.1.

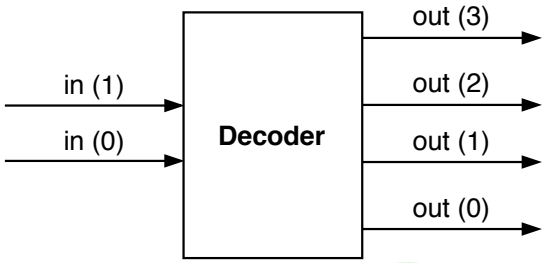


Figure 3.1: A simple 2 to 4 decoder.

```
//Example 2 to 4 decoder
import chisel3._
class LM_IO_Interface extends Bundle {
    val in  = Input(UInt(2.W))
    val out = Output(UInt(4.W))
}
class Decoder_2to4 extends Module {
    val io = IO(new LM_IO_Interface)

    when(io.in === "b00".U) {
        io.out := "b0001".U
    } .elsewhen(io.in === "b01".U) {
        io.out := "b0010".U
    } .elsewhen(io.in === "b10".U) {
        io.out := "b0100".U
    } .otherwise {
        io.out := "b1000".U
    }

}

println((new chisel3.stage.ChiselStage).emitVerilog(new Decoder_2to4()))
```

Listing 3.1: Decoder using when construct.

Similarly, it is also possible to implement an encoder, like the one shown in Figure 3.2, using when construct. Listing 3.2 illustrates the Chisel implementation for the encoder.

```
class EncoderIO extends Bundle {
    val in  = Input(UInt(4.W))
    val out = Output(UInt(2.W))
}
```

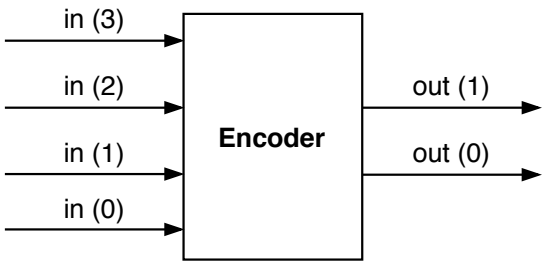


Figure 3.2: A simple 4 to 2 encoder.

```
class Encoder4to2 extends Module {
    val io = IO(new EncoderIO)

    when (io.in === "b0001".U) {
        io.out := "b00".U
    } .elsewhen(io.in === "b0010".U) {
        io.out := "b01".U
    } .elsewhen(io.in === "b0100".U) {
        io.out := "b10".U
    } .otherwise {
        io.out := "b11".U
    }
}
```

Listing 3.2: Encoder implementation using `when` construct.

3.1.2 Switch

The `switch` construct in Chisel is similar to a `case` statement in Verilog. However, there is one key difference between `switch` and `case` syntax. The `switch` construct does not have the `default` case. Due to the absence of default case, we might run into an initialization errors. In addition, `is` keyword is used to mark different cases for the `switch` construct. Chisel implementation for 2 to 4 decoder using `switch` construct is illustrated in Listing 3.3.

```
class DecoderIO extends Bundle {
    val in  = Input(UInt(2.W))
    val out = Output(UInt(4.W))
}

class Decoder2to4 extends Module {
    val io = IO(new DecoderIO)
    io.out := 0.U
    switch (io.in) {
        is ("b00".U) {
            io.out := "b0001".U
        }
        is ("b01".U) {
            io.out := "b0010".U
```

```
        }
        is ("b10".U) {
            io.out := "b0100".U
        }
        is ("b11".U) {
            io.out := "b1000".U
        }
    }
}
```

Listing 3.3: Decoder using `switch` construct.

3.2 The ALU Module

The Arithmetic Logic Unit (ALU) is one of the core building blocks of a microprocessor. We will develop an ALU that will perform all the operations required by the base instruction set architecture, RV32I [6]. An ALU requires two operands and produces the result based on the operation selected. The block diagram, illustrating the ALU interfaces is shown in Figure 3.3.

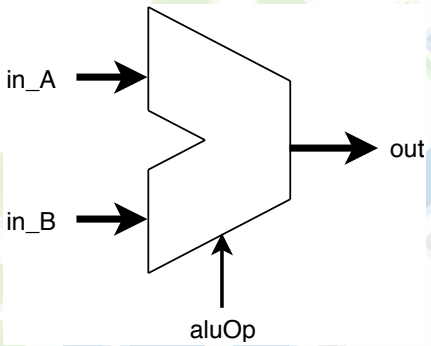


Figure 3.3: ALU block diagram representation.

3.2.1 ALU Operations

Different operations that will be performed by the ALU are grouped in the form of an `object`. A Scala `object` is used to define all the operations supported by the ALU. Listing 3.4 defines the ALUOP object for this purpose.

```
object ALUOP {
    // ALU Operations, may expand/modify in future
    val ALU_ADD     = 0.U(4.W)
    val ALU_SUB     = 1.U(4.W)
    val ALU_AND     = 2.U(4.W)
    val ALU_OR      = 3.U(4.W)
    val ALU_XOR     = 4.U(4.W)
    val ALU_SLT     = 5.U(4.W)
    val ALU_SLL     = 6.U(4.W)
    val ALU_SLTU    = 7.U(4.W)
    val ALU_SRL     = 8.U(4.W)
    val ALU_SRA     = 9.U(4.W)
```

```
    val ALU_COPY_A = 10.U(4.W)
    val ALU_COPY_B = 11.U(4.W)
    val ALU_XXX    = 15.U(4.W)
}
```

Listing 3.4: Alu operations defined as an object.

3.2.2 ALU Parameterization

Parameterization is one of the highly acclaimed features of Chisel based hardware design, providing flexibility to the design. There are many different ways to parameterize a design as we will learn in Exp 5. To parameterize the ALU design, we have chosen trait, which is a fundamental code reuse block in Scala. We will learn more about traits in Exp 13

We define trait Config, as provided by Listing 3.5, for ALU parameterization. This is a rather simple trait, which only specifies the word length and the control signal width. We use this trait by extending ALUIO bundle as well as ALU module with Config (see Listing 3.6).

```
trait Config {
    // word length configuration parameter
    val WLEN           = 32

    // ALU operation control signal width
    val ALUOP_SIG_LEN  = 4
}
```

Listing 3.5: Config trait.

It is important to notice that Listing 3.6 starts with `import` ALUOP._ despite the fact that object ALUOP is defined in the same file. Another aspect is the use of Scala wildcard "_". In the current scope, it simply means that include all the objects and classes implemented in the ALUOP package (which is object ALUOP only in this case). Scala wildcard will be discussed in detail in Exp 12.

```
import ALUOP._

class ALUIO extends Bundle with Config {
    val in_A       = Input(UInt(WLEN.W))
    val in_B       = Input(UInt(WLEN.W))
    val alu_Op     = Input(UInt(ALUOP_SIG_LEN.W))
    val out        = Output(UInt(WLEN.W))
    val sum        = Output(UInt(WLEN.W))
}

class ALU extends Module with Config {
    val io = IO(new ALUIO)

    val sum    = io.in_A + Mux(io.alu_Op(0), -io.in_B, io.in_B)
    val cmp    = Mux(io.in_A (XLEN-1) === io.in_B(XLEN-1), sum(XLEN-1),
    Mux(io.alu_Op(1), io.in_B(XLEN-1), io.in_A(XLEN-1)))
```

```
    val shamt  = io.in_B(4,0).asUInt
    val shin   = Mux(io.alu_Op(3), io.in_A, Reverse(io.in_A))
    val shiftr = (Cat(io.alu_Op(0) && shin(XLEN-1), shin).asSInt >> shamt)(
      XLEN-1, 0)
    val shiftl = Reverse(shiftr)

    val out =
    Mux(io.alu_Op === ALU_ADD.U || io.alu_Op === ALU_SUB.U, sum,
    Mux(io.alu_Op === ALU_SLT.U || io.alu_Op === ALU_SLTU.U, cmp,
    Mux(io.alu_Op === ALU_SRA.U || io.alu_Op === ALU_SRL.U, shiftr,
    Mux(io.alu_Op === ALU_SLL.U, shiftl,
    Mux(io.alu_Op === ALU_AND.U, (io.in_A & io.in_B),
    Mux(io.alu_Op === ALU_OR.U,  (io.in_A | io.in_B),
    Mux(io.alu_Op === ALU_XOR.U, (io.in_A ^ io.in_B),
    Mux(io.alu_Op === ALU_COPY_A.U, io.in_A,
    Mux(io.alu_Op === ALU_COPY_A.U, io.in_B, 0.U)))))))))

    io.out := out
    io.sum := sum
}
```

Listing 3.6: ALU implementation.

3.3 Interfaces

Chisel provides us a standard library of interfaces (facilitating interoperability of RTL) and generators for commonly used hardware blocks.

3.3.1 Valid Interface

Valid is a standard interface provided by the Chisel library. Specifically Valid is a bundle that adds valid bit to the data as shown in Figure 3.4. When valid data is put on data lines (bits) by the producer, it asserts valid bit. However, producer does not wait for the consumer in valid interface. This helps other devices to be synchronized.

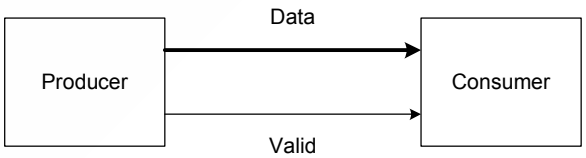


Figure 3.4: Valid signal for producer and consumer model.

Listing 3.7 shows the process of adding valid signal to the data bits, while an example illustrating the use of Valid interface is provided in Listing 3.8.

```
// data bits without valid signal
class DataWithoutValid extends Bundle {
  val data_bits = Output(UInt(8.W))
}
```

```
// data bits with valid signal
val DataWithValid = Valid(new DataWithoutValid)
```

Listing 3.7: Adding Valid interface to data.

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{
    ChiselFlatSpec,Driver,PeekPokeTester
}

class Valid_Interf extends Module{
    val io  = IO(new Bundle {
        val in  = Flipped(Valid(UInt(8.W))) //valid = Input,   bits = Input
        val out = Valid(UInt(8.W))          //valid = Output,  bits = Output
    })
    io.out := RegNext(io.in)
}
println(chisel3.Driver.emitVerilog(new Valid_Interf))
```

Listing 3.8: Valid interface illustration.

3.3.2 Decoupled Interface

Chisel provides some built-in standard interfaces that should be used whenever possible for interoperability. Decoupled is one of those standard interfaces of Bundle type, which augments ready-valid signaling to the data bits for handshaking. The ready valid signaling used by the Decoupled interface is illustrated pictorially in Figure 3.5. For decoupled IO, the producer uses the interface as is, while the consumer uses flipped interface.

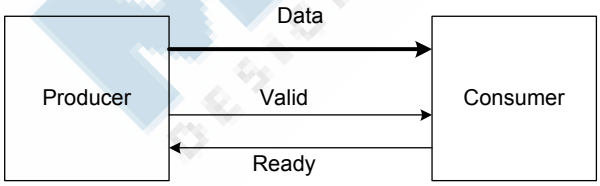


Figure 3.5: Ready valid signaling for producer consumer model used by Decoupled.

Any data type that is a subclass of the class 'Data' can be passed to the Decoupled constructor. The `object` Decoupled adds a ready-valid handshaking protocol to the data bundle. Signal 'ready' is used to show that this device is ready for communication if its high, while 'valid' is used to indicate that data 'bits' are valid. It is important to mention that no signaling requirements are imposed on ready and valid. The `apply` method of Decoupled adds the handshaking protocol using the DecoupledIO, while DecoupledIO is a subclass of ReadyValidIO signaling. Listing 3.9 illustrates the ReadyValidIO signaling. The use of type parameters (`[T]` and its variants) will be explained in Experiment 5.

```
// DecoupledIO class definition
class DecoupledIO[+T <: Data] extends ReadyValidIO[T]
```

```
class ReadyValidIO[T <: Data](gen: T) extends Bundle {
    val ready = Input(Bool())
    val valid = Output(Bool())
    val bits = Output(gen)
}
```

Listing 3.9: Ready valid interface illustration.

3.4 Exercises

Exercise 1: Refer to Listing 3.3 and implement 4 to 2 encoder using switch-is construct.

Exercise 2: Write Chisel code for a standard RISC-V ALU using switch-is construct.

3.5 Assignments

Task 1: In a standard RISC-V ALU whenever we have a branch instruction, the ALU computes it and tells the PC whether to jump or not. Implement the conditional branch module of a standard RV32I using combinational circuit and control constructs. A skeleton code is given in Listing 3.10, which can be used as starting point. Block diagram for conditional branch module is shown in Figure 3.6.

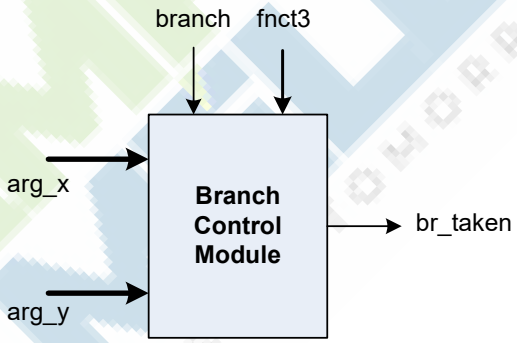


Figure 3.6: Branch control module for RV32I.

```
// Branch control (Assignment)
package lab3
import chisel3._
import chisel3.util._

class LM_IO_Interface_BranchControl extends Bundle {
    val fnct3     = Input(UInt(3.W))
    val branch    = Input(Bool())
    val arg_x     = Input(UInt(32.W))
    val arg_y     = Input(UInt(32.W))
    val br_taken  = Output(Bool())
}
```

```
class BranchControl extends Module {
    val io = IO(new LM_IO_Interface_BranchControl)
    // Start Coding here


    // End your code here
    // Well, you can actually write classes too. So, technically you have no
    limit ; )
}

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

Listing 3.10: Skeleton code for Branch control for RV32I.

Task 2: Immediate extension is an essential element of decode stage and needs to provide sign extended immediate value to the execute stage. Implement an RV32I standard immediate extension module using skeleton code available in Listing 3.11. Block level diagram of immediate extension is shown in Figure 3.7.

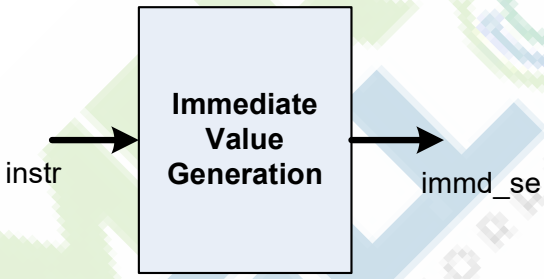


Figure 3.7: Immediate extension module for RV32I.

```
// Immediate (Assignment)
package lab3
import chisel3._
import chisel3.util._

class LM_IO_Interface_ImmdValGen extends Bundle {
    val instr = Input(UInt(32.W))
    val immd_se = Output(UInt(32.W))
}

class ImmdValGen extends Module {
    val io = IO(new LM_IO_Interface_ImmdValGen)

    // Start coding here

    // End your code here
    // Well, you can actually write classes too. So, technically you have no
    limit ; )
}
```

Listing 3.11: Skeleton code for Immediate extension module for RV32I.

Task 3: To understand the usage of Valid interface, implement 2 to 4 Decoder and wrap the output in Valid construct. Skeleton code for this task is available in Listing 3.12 and block diagram is shown in Figure 3.8.

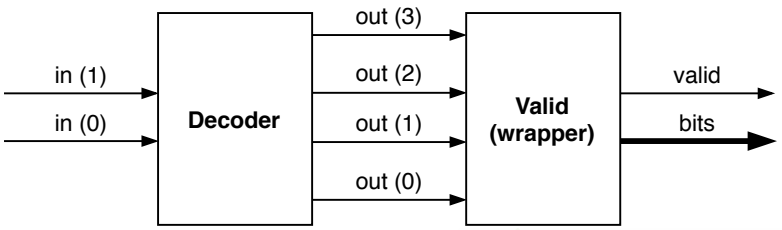


Figure 3.8: Decoder with output wrapped in Valid interface.

```
package lab3
import chisel3._
import chisel3.util._

class LM_IO_Interface_decoder_with_valid extends Bundle {
    val in  = Input(UInt(2.W))
    val out = Valid(Output(UInt(4.W)))
}

class decoder_with_valid extends Module {
    val io = IO(new LM_IO_Interface_decoder_with_valid)

    // Start coding here

    // End coding here
}
```

Listing 3.12: Skeleton code for Decoder with output wrapped in Valid interface.

Task 4: In Listing 3.6, we observe that there are two outputs of ALU, io.out and io.sum. When opcode is for ADD operation, we see that both io.out and io.sum are same. What might be the use of these two different outputs?

Experiment 4

Chisel Testers

Objective

Be familiar with the Chisel testing and learn how to write and run tests to verify proper functioning of the device-under-test (DUT).

4.1 Testing in Chisel

As is the case with almost all hardware description languages, at the end of the day, we need to verify for proper working of the hardware that has been designed. *Chisel* based hardware design is no exception. The strength of testing in *Chisel* lies in the fact that we can use all the features available in Scala to write tests.

4.1.1 Chisel Tester

Most common tool, that is used for testing in Chisel, is the iotesters. The iotesters supports following three different harnesses for testing a DUT.

1. PeekPokeTester
2. SteppedHWIOTester
3. OrderedDecoupledHWIOTester

Among the three different testing harnesses available, we will primarily focus on `PeekPokeTester`. Furthermore, there is another tester named *tester2*, which is available for testing. However, it is in the experimental phase and we will briefly touch it in the upcoming labs.

4.2 PeekPokeTester

PeekPokeTester is a mature testing methodology in *Chisel*. The library for PeekPokeTester can be imported using the following command.

```
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}
```

When performing an IO testing, we need to drive the inputs and check whether the outputs are according to our expectation. For debugging purpose, we can display the driving inputs along with corresponding outputs produced by the DUT on a terminal window. To test sequential circuits, we also need to drive the clock. The PeekPokeTester has four constructs as discussed next.

- `peek`: As the name suggests, we can lookup the value of an output using the construct `peek`. Peek returns a literal value. Peek can also be used to get the values of driving inputs. The syntax for `peek` is:

```
peek(io)
```

- `poke`: In order to drive an input of the DUT, we poke that input with a permissible value. The syntax for `poke` is:

```
poke(io, value)
```

- `expect`: To compare an output produced by the DUT with a literal value or another input/output, we can use expect. The syntax for `expect` is:

```
expect(io, equal_to)
```

- `step`: To drive the clock, we have the `step` construct. This allows us to drive the implicit clock of the module by an arbitrary number of cycles. The syntax for `step` is:

```
step(n)     // n is no. of cycles and accepts integer values
```

4.2.1 Project Directory Hierarchy

To manage a project involving multiple source files implementing different modules along with corresponding tests, Chisel uses a predefined project directory hierarchy. This structure of project directory hierarchy is shown in Figure 4.1.

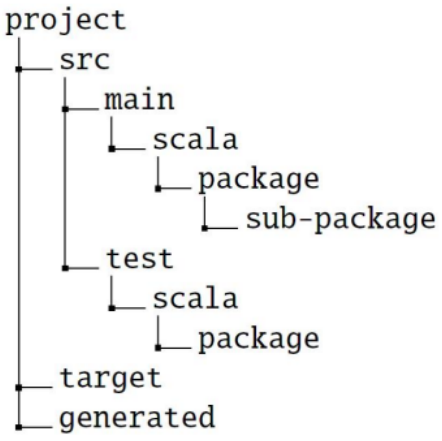


Figure 4.1: Project structure hierarchy.

4.2.2 Writing Test

We will follow learn by example methodology to be able to write tests using PeekPokeTester. For that purpose we have chosen the example of a Mux tree. The two main components of any module testing are the DUT and the Test as discussed below.

- DUT: We select a Mux tree with three inputs and two sections lines as our DUT. The implementation of this Mux is provided in Listing 4.1.

```
package LM_Chisel
import chisel3._

class MuxTreeIO extends Bundle {
    val in_1   = Input(UInt(32.W))
    val in_2   = Input(UInt(32.W))
    val in_3   = Input(UInt(32.W))
    val sel_1  = Input(Bool())
    val sel_2  = Input(Bool())
    val out    = Output(UInt())
}

// 3 to 1 MuxTree implementation
class MuxTree extends Module {
    val io = IO(new MuxTreeIO)

    // update the output
    io.out := Mux(io.sel_2, io.in_3, Mux(io.sel_1, io.in_2, io.in_1))
}
```

Listing 4.1: Mux tree with three inputs.

- Test: Writing a test requires driving of the DUT's inputs and checking the corresponding outputs. In addition, input and output values can also be printed for debugging. Listing 4.2 implements the test.

```
package LM_Chisel
import chisel3._
import chisel3.iotesters.{Driver, PeekPokeTester}

class TestMuxT(c: MuxTree) extends PeekPokeTester(c) {
    val in1  = 0x11111111
    val in2  = 0x22222222
    val in3  = 0x33333333

    poke(c.io.in_1, in1.U)
    poke(c.io.in_2, in2.U)
    poke(c.io.in_3, in3.U)

    poke(c.io.sel_1, false.B)
    poke(c.io.sel_2, false.B)
    expect(c.io.out, in1.U)
    step(1)

    poke(c.io.sel_1, true.B)
    poke(c.io.sel_2, false.B)
```

```
    expect(c.io.out, in2.U)
    step(1)

    poke(c.io.sel_1, true.B)
    poke(c.io.sel_2, true.B)
    expect(c.io.out, in3.U)
    step(3)
}
```

Listing 4.2: Tester for the DUT.

From Listing 4.2 we observe that the class is inherited from `PeekPokeTester`. The parameter passed to the 'TestMuxT' class is of type DUT, i.e. of type 'MuxTree'. It is worth mentioning that the DUT file is placed in "\scr \main \scala", while test file is placed in "\scr \test \scala". Observe that the first line in both the files is package LM_Chisel, which puts both the files in the same package.

4.2.3 Running Test

To run the test, we need to either create a main method in the object or extend the object from the class `App`. Our test initializer is implement by extending an object from the class `App` and is given in Listing 4.3. This test initializer code in appended to the test file.

```
// object for tester class
object MuxT_Main extends App {
    iotesters.Driver.execute(Array("--is-verbose", "--generate-vcd-output",
    "on"), () => new MuxTree) {
        c => new TestMuxT(c)
    }
}
```

Listing 4.3: Initializer for the tester.

In Listing 4.3, observe the use of configuration array to pass user options to configure the tools. For instance, we use `"--is-verbose"` to turn on the verbose mode. We can run this test, using command line or an sbt shell. An sbt shell can be opened by typing sbt in the command terminal and hitting enter. When the sbt shell opens, one can run the test using the following syntax.

```
test:runMain packageName.objectName
```

For our specific example, use the following command.

```
test:runMain LM_Chisel.MuxT_Main
```

4.2.4 Tools Invoked while Testing

When a test is run by the user command **test:run**, different tools are invoked at different stages as listed below. Further details regarding tools usage can be found in Appendix A.

- At first step the tester (PeekPokeTester in this case) is invoked
- Tester invokes chisel3 to generate the circuit
- The chisel3 invokes firrtl to compile the circuit into low firrtl
- The low firrtl invokes the firrtl-interpreter to execute the test on the DUT

4.2.5 Viewing Output

When the test is run, we can view the driving inputs and corresponding outputs in the terminal window. This detailed information is printed to the terminal (as depicted in Figure 4.2) because the verbose mode was turned on. A .vcd file is also generated due to the configuration options. One can open the .vcd file using a waveform viewer application. Figure 4.3 shows the signals for the 'MuxTree' testing.

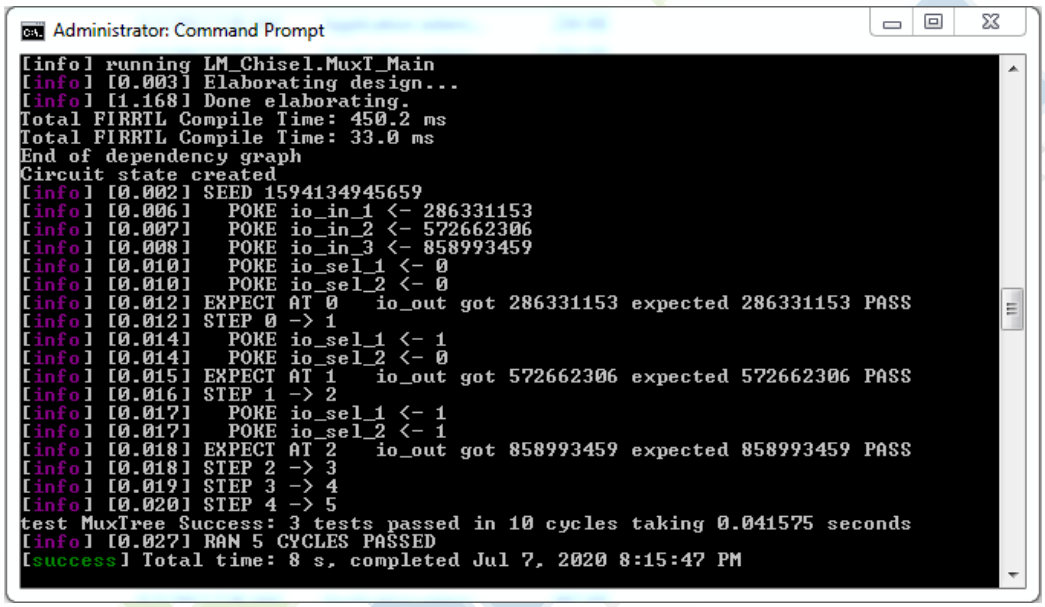


Figure 4.2: Command line output in verbose mode.

4.2.6 Tester Configurations

Apart from the few configurations that we have used so far, there are many other user configurations available. Some of the useful configurations are listed in Listing 4.4.

```
                    Selected Tester Options
                    -----------------------

-tbn, --backend-name <firrtl|treadle|verilator|ivl|vcs>
                    backend to use with tester, default is treadle

-tiv, --is-verbose    set verbose flag on PeekPokeTesters, default is false

-twffn, --wave-form-file-name <value>
                    wave form file name

-tts, --test-seed <value>
                    provides a seed for random number generator
```

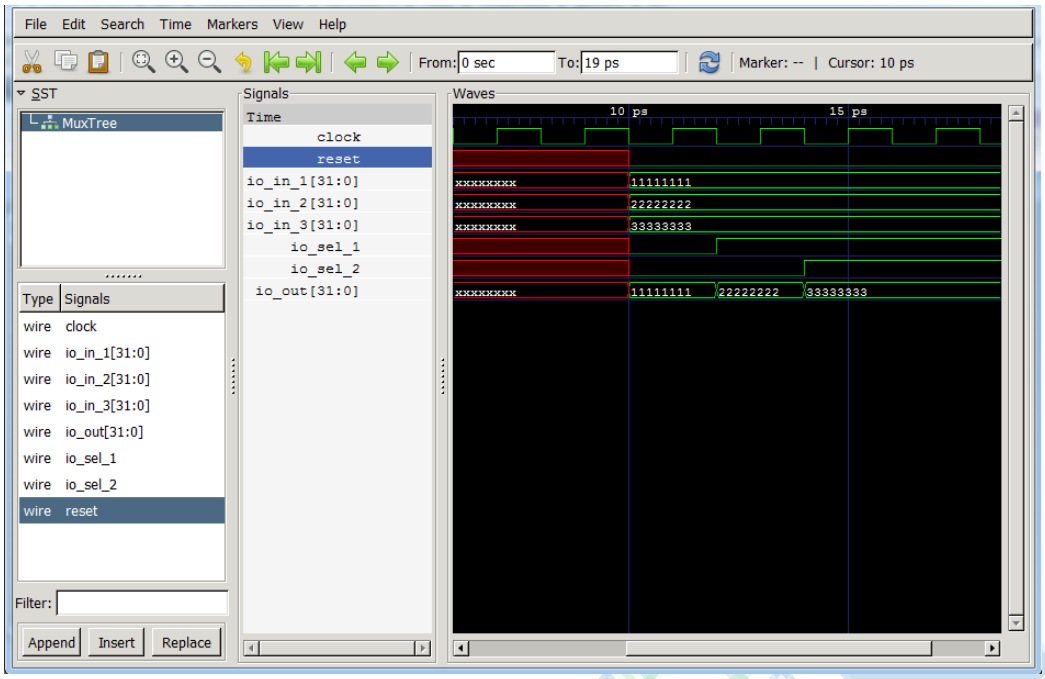


Figure 4.3: Viewing input/output signals using GTKWave.

```
-tgvo, --generate-vcd-output <value>
                        set this flag to "on" or "off", otherwise it defaults to on
                        for verilator, off for scala backends

-td <target-directory>, --target-dir <target-directory>
                        defines a work directory for intermediate files, default is
                        current directory
```

Listing 4.4: Selected user configurations.

Furthermore, to print the Verilog generated in a terminal window, we can simply add the following command to the object extending from the App.

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

To dump the verilog and FIRRTL output, one can use the corresponding options as described below.

```
chisel3.Driver.execute(Array("--target-dir","RTL_files"), () => new DUT)
```

4.3 ALU Tester

We have already implemented the ALU in Experiment 4. Now we are going to write a test, which will randomly poke different operations implemented by the ALU and will verify the results for correctness. Listing 4.5 implements random testing of the ALU and the test results are depicted in Figure 4.4

```
package LM_Chisel
```

```
import chisel3._
import chisel3.util
import chisel3.iotesters.{
    Driver, PeekPokeTester
}
import scala.util.Random
import ALUOP._

class TestALU(c: ALU) extends PeekPokeTester(c) {
    // ALU operations
    val array_op = Array(ALU_ADD , ALU_SUB, ALU_AND, ALU_OR, ALU_XOR,ALU_SLT, ALU_SLL
    , ALU_SLTU, ALU_SRL, ALU_SRA, ALU_COPY_A, ALU_COPY_B, ALU_XXX)

    for (i <- 0 until 100) {
        val src_a  = Random.nextLong()& 0xFFFFFFFFL
        val src_b = Random.nextLong()& 0xFFFFFFFFL
        val opr = Random.nextInt(12)
        val aluop  = array_op(opr)

        // ALU functional implementation using Scala match
        val result = aluop match {
            case ALU_ADD  => src_a + src_b
            case ALU_SUB  => src_a - src_b
            case ALU_AND  => src_a & src_b
            case ALU_OR   => src_a | src_b
            case ALU_XOR  => src_a ^ src_b
            case ALU_SLT  => (src_a.toInt < src_b.toInt).toInt
            case ALU_SLL  => src_a  << (src_b & 0x1F)
            case ALU_SLTU => (src_a < src_b).toInt
            case ALU_SRL  => src_a >> (src_b & 0x1F)
            case ALU_SRA  => src_a.toInt >> (src_b & 0x1F)
            case ALU_COPY_A => src_a
            case ALU_COPY_B => src_b
            case _        => 0
        }

        val result1: BigInt = if (result < 0)
        (BigInt(0xFFFFFFFFL)+result+1) & 0xFFFFFFFFL
        else result & 0xFFFFFFFFL

        poke(c.io.in_A, src_a.U)
        poke(c.io.in_B, src_b.U)
        poke(c.io.alu_Op, aluop)
        step(1)
        expect(c.io.out, result1.asUInt)
    }
    step(2)
}

// object for tester class
object ALU_Main extends App {
    iotesters.Driver.execute(Array("--is-verbose", "--generate-vcd-output", "on", "--
     backend-name", "firrtl"), () => new ALU) {
```

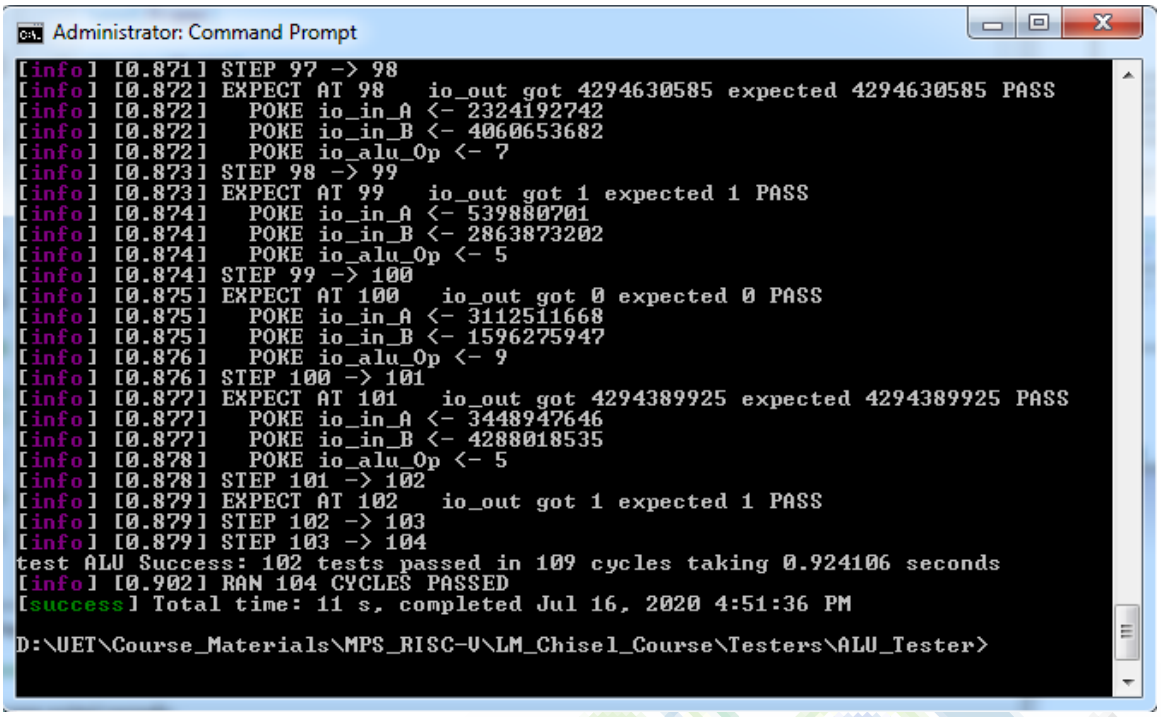


Figure 4.4: ALU test results.

```
        c => new TestALU(c)
    }
}
```

Listing 4.5: ALU tester implementation.

4.4 Exercises

Exercise 1: Extend the ALU tester that tests operations not supported by the ALU and outputs the user defined illegal result.

4.5 Assignments

Task 1: Write the test for Branch module.

Task 2: Write the test for Immediate generation module.

Task 3: You are provided with a code for a buggy ALU. Test it and find out the bug(s).

Experiment 5

Parameterization

Objective

Learn different techniques for parameterized hardware generation to achieve design flexibility and reusability.

5.1 Parameterization in Chisel

Parameterization is a powerful Scala feature that is readily available in Chisel to develop flexible hardware generators. From a given hardware generator, we can realize many variants by choosing a set of parametric values. In other words, by varying parametric values, we can realize a different hardware from the parameterized generator. Broadly speaking, hardware generation can involve parameterization in one or combination of the following scenarios.

- Bitwidth parameterization
- Functions with type parameters
- Modules with type parameters
- Bundle parameterization

5.1.1 Bitwidth Parameterization

Bitwidth parameterization is the simplest one, where we can parameterize the widths for the IOs. Parameters specifying bitwidths can be passed as arguments to the constructor of the class. During compilation the constructor will be called and it sets the widths for the IO bundle. Listing 5.1 implements an ALU module where the width of IOs is parameterized by passing parameters as an argument to ALU class constructor. It is important to note that the width parameter is of type integer. All parameters passed as an argument have some type associated with them.

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{Driver,PeekPokeTester}

class ALU(width_parameter:Int) extends Module {
  val io = IO(new IO_Interface(width_parameter))

  io.alu_out := 0.U
  val index = log2Ceil(width_parameter)
```

```
  switch(io.alu_oper) { //AND
    is("b0000".U) {
      io.alu_out := io.arg_x & io.arg_y
    } //OR
    is("b0001".U) {
      io.alu_out := io.arg_x | io.arg_y
    } //ADD
    is("b0010".U) {
      io.alu_out := io.arg_x + io.arg_y
    } //SUB
    is("b0110".U) {
      io.alu_out := io.arg_x - io.arg_y
    } //XOR
    is("b0011".U) {
      io.alu_out := io.arg_x ^ io.arg_y
    } //SLL
    is("b0100".U) {
      io.alu_out := io.arg_x << io.arg_y(index-1, 0)
    } //SRL
    is("b0101".U) {
      io.alu_out := io.arg_x >> io.arg_y(index-1, 0)
    } //SRA
    is("b0111".U) {
      io.alu_out := (io.arg_x.asSInt >> io.arg_y(index-1, 0)).asUInt
    } //SLT
    is("b1000".U) {
      io.alu_out := io.arg_x.asSInt < io.arg_y.asSInt
    } //SLTU
    is("b1001".U) {
      io.alu_out := io.arg_x < io.arg_y
    }
  }
}

class IO_Interface(width:Int) extends Bundle {
  val alu_oper = Input(UInt(width.W))
  val arg_x = Input(UInt(width.W))
  val arg_y = Input(UInt(width.W))
  val alu_out = Output(UInt(width.W))
}

println((new chisel3.stage.ChiselStage).emitVerilog(new ALU(32)))
println((new chisel3.stage.ChiselStage).emitVerilog(new ALU(64)))
```

Listing 5.1: Parameterized ALU.

5.1.2 Functions with Type Parameters

There are many ways to define a function in Scala but in all cases, the type of the inputs to the function must be defined while in some cases the return type is optional, if the return type is not

defined then it will be inferred by the compiler.

It is possible to parameterize functions with types in Chisel. Observe Listing 5.2, where `'T'` in the expression `'[T <: Data]'` defines a *Type Parameter*. Data is the root class of the type system in Chisel. The type 'T' can be of any type of subclass of 'Data' e.g. `UInt`, `SInt`, Vec, etc. This gives us the flexibility to write generic code and configure its inputs or outputs ports type during instantiation.

In Listing 5.2, we can modify the type of input and output ports. Any type of subclass of 'Data' can be used. So we have a multiplexer that can accept any types for multiplexing.

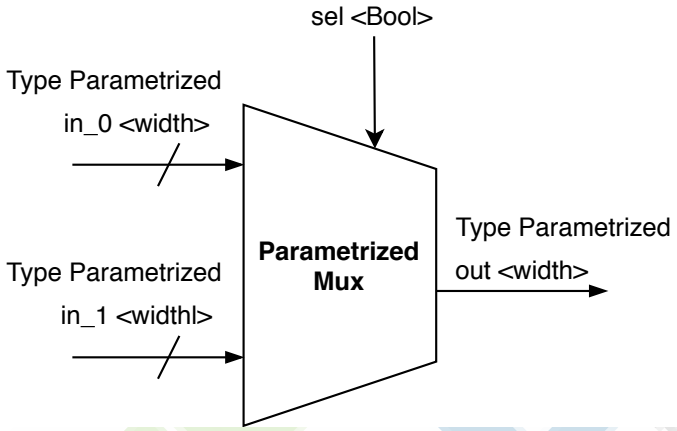


Figure 5.1: Parameterized Mux.

.

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

class eMux[T<:Data](gen:T) extends Module{
    val io = IO(new Bundle{
        val out = Output(gen)
        val in1 = Input(gen)
        val in2 = Input(gen)
        val sel = Input(Bool())
    })
    io.out := Mux2_to_1(io.in2, io.in1, io.sel)

    def Mux2_to_1[T <: Data](in_0:T, in_1:T, sel:Bool):T = {
        Mux(sel, in_1, in_0)
    }
}
println((new chisel3.stage.ChiselStage).emitVerilog(new eMux(SInt(2.W))))
```

Listing 5.2: Parameterized Mux.

5.1.3 Modules with Type Parameters

The data type of module ports can also be parameterized. Here we can also perform type parameterization using T-type. It's the same as discussed in the last section. To illustrate type parameters we use a 2x2 crossbar switch as shown in Figure 5.2 and its implementation is provided in Listing 5.3.

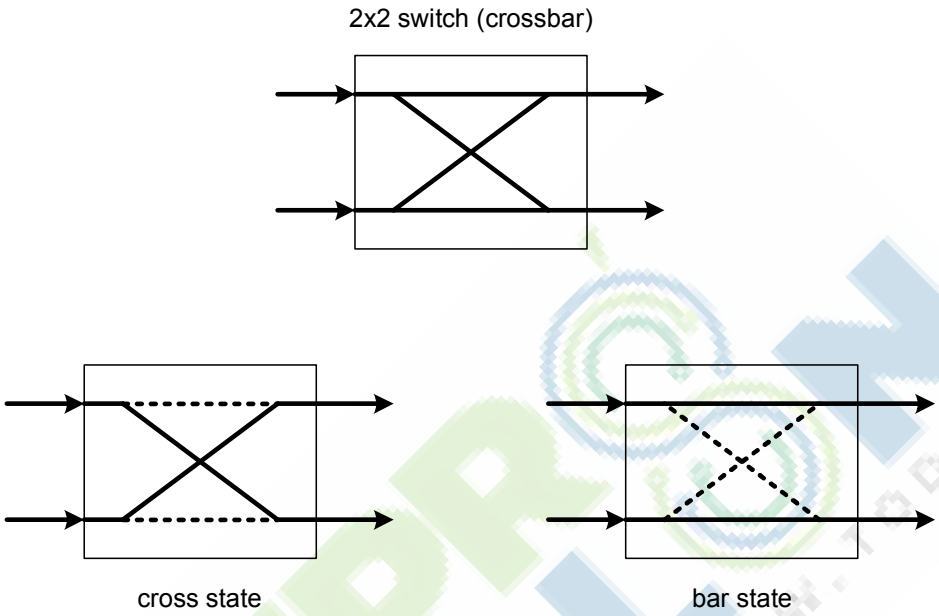


Figure 5.2: Network switch (2x2).

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

class switch_2cross2 [T <: Data](parameter:T) extends Module{
    val io = IO(new Bundle{
        val in1  = Input(parameter)
        val in2  = Input(parameter)
        val out1  = Output(parameter)
        val out2  = Output(parameter)
        val sel   = Input(Bool())
    })

    when(io.sel){
        io.out1 := io.in2
        io.out2 := io.in1
    }
    .otherwise{
        io.out1 := io.in1
        io.out2 := io.in2
    }
}

println(chisel3.Driver.emitVerilog(new switch_2cross2(UInt(8.W))))
```

Listing 5.3: Parameterized 2x2 switch.

5.1.4 Bundle Parameterization

To parameterize bundles, all parameters must be passed when an object of a subclass is created. If the bundle is declared within the same class in which arguments have been passed then it's simple but if they are declared in a separate class then arguments must be passed to the object of a subclass which extends the Bundle class.

Listing 5.4 illustrates the use of bundle parameterization. The bundle in Listing 5.4 has a parameter of type `T`, which is a subtype of Chisel's `Data` type. In the bundle, we define different fields by invoking cloneType on the parameter.

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

class IO_Interface[T <: Data] (data_type:T) extends Bundle{
    val in1  = Input(data_type.cloneType)
    val in2  = Input(data_type.cloneType)
    val out  = Output(data_type.cloneType)
    val sel  = Input(Bool())
}

class Adder(size: UInt) extends Module{
    val io = IO(new IO_Interface(size))

    io.out := io.in1 + io.in2
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Adder(15.U)))
```

Listing 5.4: Bundle parametrization.

5.1.5 Chisel Method: cloneType

Constructing copies of bundles for various purposes requires cloning. Chisel can automatically figure out how to clone bundles in most cases. For some parameterized bundles, Chisel may not automatically figure out how to clone. Solution to this problem is to create a custom `cloneType` method in the parameterized bundle. This is illustrated in Listing 5.5.

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

class Adder_Inputs(x: Int, y: Int) extends Bundle {
    val in1 = UInt(x.W)
    val in2 = UInt(y.W)

    // creating a custom cloneType method
```

```
    override def cloneType = (new Adder_Inputs(x, y)).asInstanceOf[this.type]
}

class Adder(inBundle: Adder_Inputs, outSize: Int) extends Module {
    val io = IO(new Bundle {
        val out = Output(UInt(outSize.W))

        // chiselTypeOf returns the chisel type of the object
        val in_bundle = Input(chiselTypeOf(inBundle))
    })
    io.out := io.in_bundle.in1 + io.in_bundle.in2
}

class Top(in1Size: Int, in2Size: Int, outSize: Int) extends Module {
    val io = IO(new Bundle {
        val out = Output(UInt(outSize.W))
        val in = Input(UInt(in1Size.W))
    })

    // input bundle instance
    val inBundle = Wire(new Adder_Inputs(in1Size, in2Size))
    inBundle := DontCare

    // module instance
    val m = Module(new Adder(inBundle, outSize))
    m.io.in_bundle.in1 := io.in
    m.io.in_bundle.in2 := io.in
    io.out := m.io.out
}

println((new ChiselStage).emitVerilog(new Top(18, 30, 32)))
```

Listing 5.5: Use of cloneType in Chisel.

5.2 Advanced Parameterization

We discuss two instances of advanced parameterization here. The first scenario is where we use a class as parameter. Listing 5.6 illustrates how an instance of class Parameters can itself be passed as a parameter.

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

class Parameters(dWidth: Int, aWidth: Int) extends Bundle{
    val addrWidth = UInt(aWidth.W)
    val dataWidth = UInt(dWidth.W)
}

class DataMem (params: Parameters) extends Module {
```

```
    val io = IO(new Bundle{
        val data_in  = Input(params.dataWidth)
        val data_out = Output(params.dataWidth)
        val addr     = Input(params.addrWidth)
        val wr_en    = Input(Bool())
    })

    // Make memory of 32 x 32
    val memory = Mem(32, params.dataWidth)

    io.data_out := 0.U

    when(io.wr_en) {
        memory.write(io.addr, io.data_in)
    } .otherwise {
        io.data_out := memory.read(io.addr)
    }
}

val params = (new Parameters(32, 5))
println((new chisel3.stage.ChiselStage).emitVerilog(new DataMem(params)))
```

Listing 5.6: Advanced parametrization using class instance as parameter.

From the second illustration given in Listing 5.7, we observe that there are multiple parameter lists, `(n: Int, generic: T)` and `(op: (T, T) => T)`. The argument, n: Int, in the first parameter list is a simple integer parameter, second argument, generic: T, is the generic type T parameter, and is a subtype of Data.

The second parameter list i.e. `(op: (T, T) => T)` is a function. Since Scala is functional programming, functions are treated as values in Scala, and hence can become arguments. The functional mapping `(T, T) => T` represents a function with two arguments of type T and returns an output of type T. Recall `T` is a subclass of `Data`. Since op is in second parameter list, which requires to infer T from generic, and then use it for op. A higher order method (.reduce) is used to apply the operator on all values and produce an output (single value). The function parameter is realized for addition and AND operations.

```
import chisel3._

// class definition with function as parameter
class Operator[T <: Data](n: Int, generic: T)(op: (T, T) => T) extends
    Module {
    require(n > 0) // "reduce only works on non-empty Vecs"

    val io = IO(new Bundle {
        val in = Input(Vec(n, generic))
        val out = Output(generic)
    })
```

```
    io.out := io.in.reduce(op)
}

// Implement addition operation
object UserOperator1 extends App {
    println((new chisel3.stage.ChiselStage).emitVerilog(new Operator(2, UInt
    (16.W))(_ + _)))
}

// Implement AND operation
object UserOperator2 extends App {
    println((new chisel3.stage.ChiselStage).emitVerilog(new Operator(3, UInt
    (8.W))(_ & _)))
}
```

Listing 5.7: Advanced parametrization with operator parameter.

5.3 Illustration from Rocket Chip

The parameterization example is based on the ALU implementation. The data width for ALU is parameterized using implicit parameter xLen as can be viewed from Listing 5.8. We will discuss more about implicit parameters later.

```
class ALU(implicit p: Parameters) extends CoreModule()(p) {
    val io = new Bundle {
        val dw = Bits(INPUT, SZ_DW)
        val fn = Bits(INPUT, SZ_ALU_FN)
        val in2 = UInt(INPUT, xLen)  // xLen is an implicit parameter
        val in1 = UInt(INPUT, xLen)
        val out = UInt(OUTPUT, xLen)
        val adder_out = UInt(OUTPUT, xLen)
        val cmp_out = Bool(OUTPUT)
    }

    // ADD, SUB
    val in2_inv = Mux(isSub(io.fn), ~io.in2, io.in2)
    val in1_xor_in2 = io.in1 ^ in2_inv
    io.adder_out := io.in1 + in2_inv + isSub(io.fn)

    // SLT, SLTU
    val slt =
    Mux(io.in1(xLen-1) === io.in2(xLen-1), io.adder_out(xLen-1),
    Mux(cmpUnsigned(io.fn), io.in2(xLen-1), io.in1(xLen-1)))
    io.cmp_out := cmpInverted(io.fn) ^ Mux(cmpEq(io.fn), in1_xor_in2 === UInt
      (0), slt)

    // SLL, SRL, SRA
    val (shamt, shin_r) =
    if (xLen == 32) (io.in2(4,0), io.in1)
```

```
    else {
        require(xLen == 64)
        val shin_hi_32 = Fill(32, isSub(io.fn) && io.in1(31))
        val shin_hi = Mux(io.dw === DW_64, io.in1(63,32), shin_hi_32)
        val shamt = Cat(io.in2(5) & (io.dw === DW_64), io.in2(4,0))
        (shamt, Cat(shin_hi, io.in1(31,0)))
    }
    val shin = Mux(io.fn === FN_SR  || io.fn === FN_SRA, shin_r, Reverse(
      shin_r))
    val shout_r = (Cat(isSub(io.fn) & shin(xLen-1), shin).asSInt >> shamt)(
      xLen-1,0)
    val shout_l = Reverse(shout_r)
    val shout = Mux(io.fn === FN_SR || io.fn === FN_SRA, shout_r, UInt(0)) |
    Mux(io.fn === FN_SL,                       shout_l, UInt(0))

    // AND, OR, XOR
    val logic = Mux(io.fn === FN_XOR || io.fn === FN_OR, in1_xor_in2, UInt(0)
      ) |
    Mux(io.fn === FN_OR || io.fn === FN_AND, io.in1 & io.in2, UInt(0))
    val shift_logic = (isCmp(io.fn) && slt) | logic | shout
    val out = Mux(io.fn === FN_ADD || io.fn === FN_SUB, io.adder_out,
      shift_logic)

    io.out := out
    if (xLen > 32) {
        require(xLen == 64)
        when (io.dw === DW_32) {
            io.out := Cat(Fill(32, out(31)), out(31,0))
        }
    }
}
```

Listing 5.8: The ALU implementation in Rocket core.

5.4 Exercises

Exercise 1: Refer to the Listing 5.2, Type parametrize the Mux using bundles only.

Exercise 2: Refer to the Listing 5.7, modify the logic, make a vector of output and apply op on each input and assign resultant bits to corresponding outputs.

Exercise 3: Refer to the Listing 5.5. What happens if we don't use cloneType? If a problem arises due to this change, what will be it's solution?

5.5 Assignments

Task 1: Write Chisel code for the parameterized AHB lite interconnect which can connect 1 master to n number of slaves. For further detail refer to AHB 3 lite spec document [1].

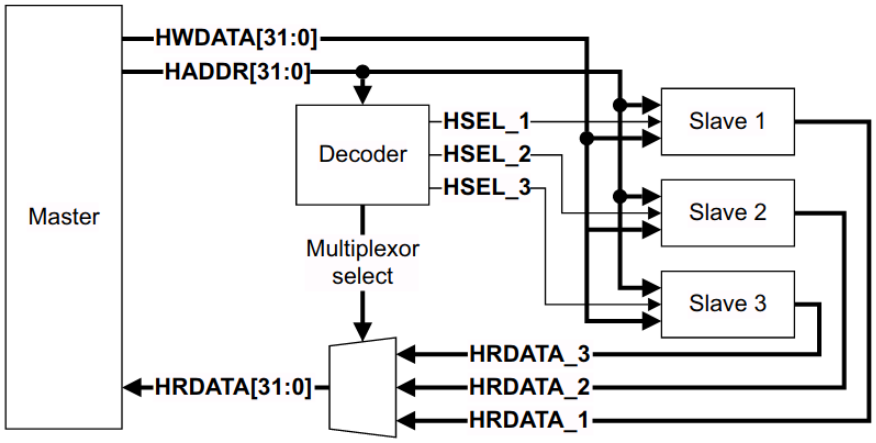


Figure 5.3: AHB lite interconnect [1].

Hint: You can create an object with vector of Address space of slave and use in address decoder. And create a bundle of AHB signals.

Task 2: Make an eMux with inclusive typecasting. Use different types for the two inputs and explain your analysis and findings based on the type of the output.

Experiment 6

Sequential Circuits

Objective

Learn the implementation of new sequential circuits as well as usage of existing modules (Chisel utilities).

6.1 Register Modules

Sequential circuits are heart of any digital design. They are used to implement states and state elements. In addition, state machines and memories, as discussed in subsequent experiments, can be constructed from sequential circuits. There are three different constructs in Chisel to define registers as will be discussed next.

6.1.1 Reg

A simple D type flip flop is used to define `Reg` construct in *Chisel*. Figure 6.1 shows block diagram for single- and multi- bit register using internal clock.

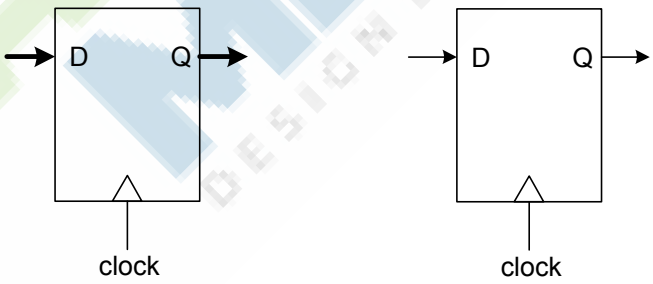


Figure 6.1: One or multi-bit register

The syntax for realizing an n bit register using `Reg` is given below.

```
val reg_nBit = Reg(UInt(n.W))
```

Using the above syntax we have defined a register with signal type `UInt` and having n-bits width. The output of `Reg` is updated with the input value on positive edge of the clock and effectively is delayed by at most one clock cycle. One key attribute of `Reg` is that we cannot assign an initial value to it. Rather the compiler will initialize it with a random value.

6.1.2 RegInit

To overcome the limitation of Reg, which does not allow to assign an initial value, we can use `RegInit` construct to assign an initial value on reset. The block diagram of `RegInit` is shown below in Figure 6.2. From Figure 6.2, we can notice that the only difference between Reg and RegInit is the presence of a Mux, on input signal path, in case of RegInit. When the *reset* is asserted, init_value is applied to the register input.

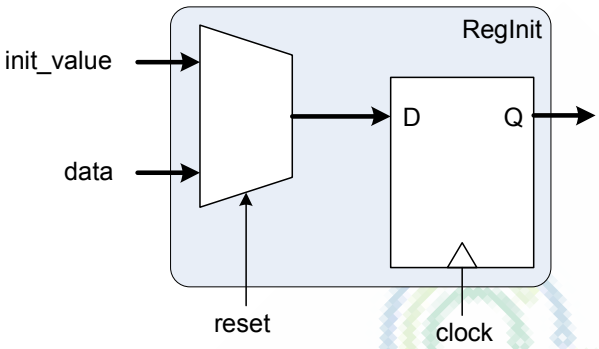


Figure 6.2: Block diagram of `RegInit` construct.

The syntax to instantiate a register using `RegInit` is given below, while Listing 6.1 provides few illustrations for the usage of `RegInit`.

```
val reg_withInit = RegInit(initial_value(n.W))
```

```
// following uses of RegInit are valid
val reg1 = RegInit(24.U(8.W))
val reg2 = Reg(UInt(8.W))
val reg3 = RegInit(reg2)

// following uses are invalid
val reg1 = RegInit(0.U(UInt(8.W)))
val reg2 = RegInit(UInt(8.W))
```

Listing 6.1: Illustration of `RegInit`.

6.1.3 RegNext

`RegNext` is another construct from *Chisel* library, which allows to assign an initial value to the register and one can also connect both the input and the output in one go. It can be used to get one cycle delayed version of the input signal. Connecting an input as well as an output can be done using `RegNext` construct as follows.

```
io.out := RegNext(io.in)
```

6.1.4 RegEnable

Another useful construct is `RegEnable`, which allows to control the updating of the register output as shown in Figure 6.3. The following syntax illustrates its usage.

```
val regWithEnable = RegEnable(nextVal, ena)
```

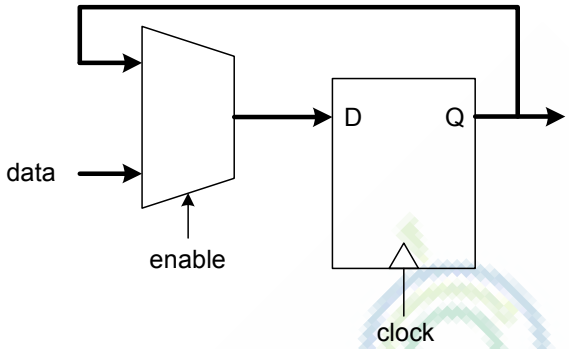


Figure 6.3: Register with enable input.

6.2 Shift Register

Now with the information we have acquired lets make a simple shift register, with serial in and parallel out connectivity as shown in Figure 6.4. Listing 6.2 implements a simple shift-register with 4-bit parallel output.

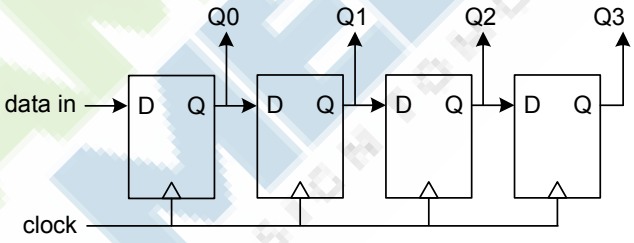


Figure 6.4: Shift register with 4-bit parallel output.

```
// shift register example
import chisel3._

class shift_register(val init: Int = 1) extends Module {
    val io = IO(new Bundle{
        val in = Input(Bool())
        val out = Output(UInt(4.W))
    })
    val state = RegInit(init.U(4.W))    // register initialization

    // serial data in at LSB
    val nextState = (state << 1) | io.in
    state := nextState
```

```
    io.out := state
}
println((new chisel3.stage.ChiselStage).emitVerilog(new shift_register))
```

Listing 6.2: Shift register

6.3 Counter

To this point we have been through many variants of simple counter implementation. Now we are going to implement rather a bit sophisticated variant of counter that can count using predefined minimum and maximum values, while optimizing (minimizing) for the hardware resources. The implementation is shown below in Listing 6.3.

```
// Optimized counter example
import chisel3._
import chisel3.util._

class counter(val max: Int, val min: Int = 0) extends Module {
    val io = IO(new Bundle{
        val out = Output(UInt(log2Ceil(max).W))
    })
    val counter = RegInit(min.U(log2Ceil(max).W))

    // If the max count is of power 2 and the min value = 0,
    // then we can skip the comparator and the Mux
    val count_buffer = if (isPow2(max) && (min == 0)) counter + 1.U
    else Mux(counter === max.U, min.U, counter + 1.U)
    counter := count_buffer
    io.out  := counter
}
println((new chisel3.stage.ChiselStage).emitVerilog(new counter(32)))
```

Listing 6.3: Counter implementation using minimum hardware resources.

In the above example, if the maximum count value is equal to some power of 2 while the minimum value is 0 then we can omit the comparator and Mux. For this special case, we can use a simple incrementing register that overflows when it reaches its maximum value. The conditional code in Listing 6.3 can also be written as follows.

```
val count_buffer = Mux(isPow2(max)&&(min==0).B, counter+1.U, Mux(counter ===
    max.U, min.U, counter+1.U) )
```

Listing 6.4: Modified condition code for counter.

Here the condition for the first Mux will be evaluated as a binary value, which will be responsible for the selection of corresponding path.

6.3.1 One-shot Timer

One shot timer is a simple clock counter that gives a high signal for one cycle when it counts to zero starting with a value from reload register. Listing 6.5 provides one possible implementation.

```
// one shot timer implementation
val timer_count = RegInit(0.U(8.W))
val done = timer_count === 0.U
val next = WireInit(0.U)

when (reload){
    next := din                 // load the data from input
}
.elsewhen (!done){
    next := timer_count - 1.U   // decrement the timer
}
timer_count := next             // update the timer
```

Listing 6.5: One-shot timer implementation.

6.3.2 PWM Generation

PWM generator requires two parameters for proper functioning. One parameter, the *max-cycle* count, determines the frequency of PWM signal and the second parameter, the duty-cycle, determines the pulse width of PWM signal. PWM generator can be implemented using the block diagram shown in Figure 6.5. The implementation of PWM generator is provided in Listing 6.6.

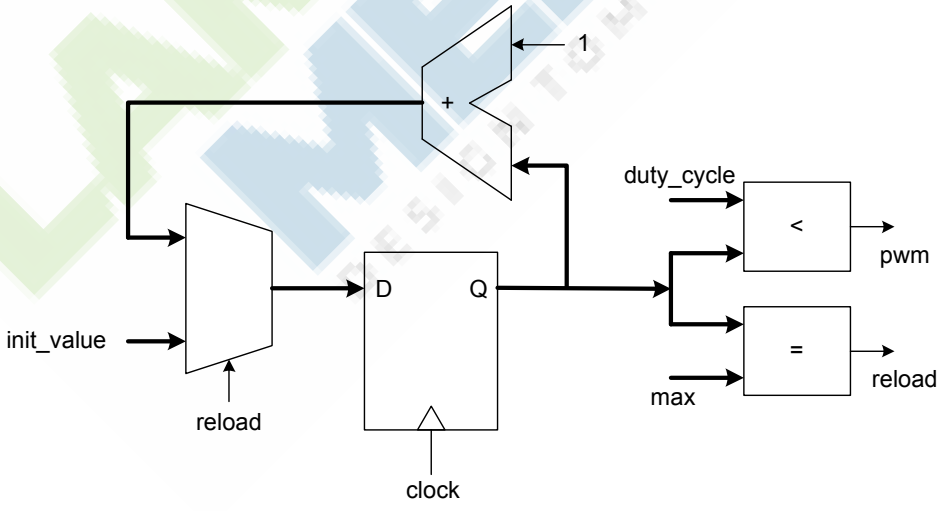


Figure 6.5: Block diagram for PWM generation.

```
// PWM example
import chisel3._
import chisel3.util._

class PWM(val max: Int = 2, val duty_cycle: Int = 1) extends Module {
    val io = IO(new Bundle{
```

```
        val out = Output(Bool())
    })
    val counter = RegInit(0.U(log2Ceil(max).W))
    counter := Mux(counter === max.U, 0.U, counter+1.U)
    io.out := duty_cycle.U > counter
}

println((new chisel3.stage.ChiselStage).emitVerilog(new PWM(15)))
```

Listing 6.6: PWM generator.

6.4 Register File

Register file is an essential building block of a microprocessor. The IO interface of a register file mainly depends on the instruction set architecture. For simple assembly instructions, employing register-register operand architecture, the processor might need two source operands from the register file and can also store the result of an operation to the register file. As a result, the register file requires two read ports and one write port. Figure 6.6 shows the block diagram of the register file with read and write ports.

A register file can be implemented using either a register vector or a memory block. Here we will implement a register file using register vector as provided in Listing 6.7.

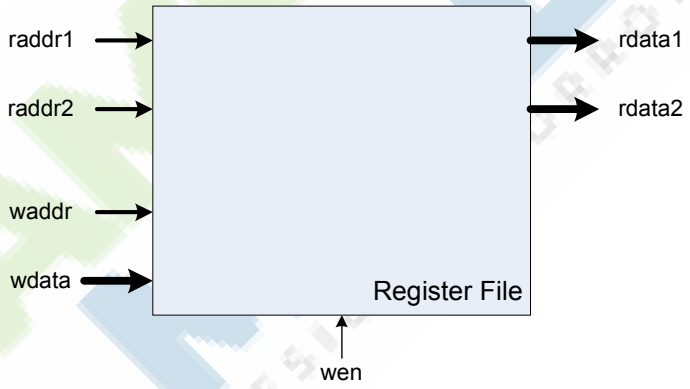


Figure 6.6: The register file with two read and one write port.

```
import chisel3._

class RegFileIO  extends Bundle with Config {
    val raddr1 = Input(UInt(5.W))
    val raddr2 = Input(UInt(5.W))
    val rdata1 = Output(UInt(XLEN.W))
    val rdata2 = Output(UInt(XLEN.W))
    val wen    = Input(Bool())
    val waddr  = Input(UInt(5.W))
    val wdata  = Input(UInt(XLEN.W))
}

class RegFile extends Module with Config {
```

```
    val io = IO(new RegFileIO)
    val regs = Reg(Vec(REGFILE_LEN, UInt(XLEN.W)))

    io.rdata1 := Mux((io.raddr1.orR), regs(io.raddr1), 0.U)
    io.rdata2 := Mux((io.raddr2.orR), regs(io.raddr2), 0.U)

    when(io.wen & io.waddr.orR) {
        regs(io.waddr) := io.wdata
    }
}
```

Listing 6.7: Register file implementation using register vector.

6.5 Pipes and Queues

6.5.1 Pipe

Pipe is a Chisel construct that can delay the inputs by a specified amount of time. It requires two input arguments, a Valid interface for data input and an integer by which latency of pipe will be set. Output data will be delayed by the specified number of clock cycles. Listing 6.8 illustrates an example implementation using Pipe construct.

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{
    ChiselFlatSpec,Driver,PeekPokeTester
}

class Pipe extends Module{
    val io  = IO(new Bundle {
        val in  = Flipped(Valid(UInt(8.W))) //valid = Input,   bits = Input
        val out = Valid(UInt(8.W))          //valid = Output,  bits = Output
    })
    io.out := Pipe(io.in,5)
}
println(chisel3.Driver.emitVerilog(new Pipe))
```

Listing 6.8: Pipe construct.

6.5.2 Queues

Queue creates a FIFO (first-in, first-out) with Decoupled interfaces for both input and output connectivity as depicted in Figure 6.7. Both the data type and the depth of the queue i.e the number of data elements, are configurable. The syntax of the Queue construct in Chisel is illustrated in Listing 6.9.

```
class My_Queue extends Module{
```

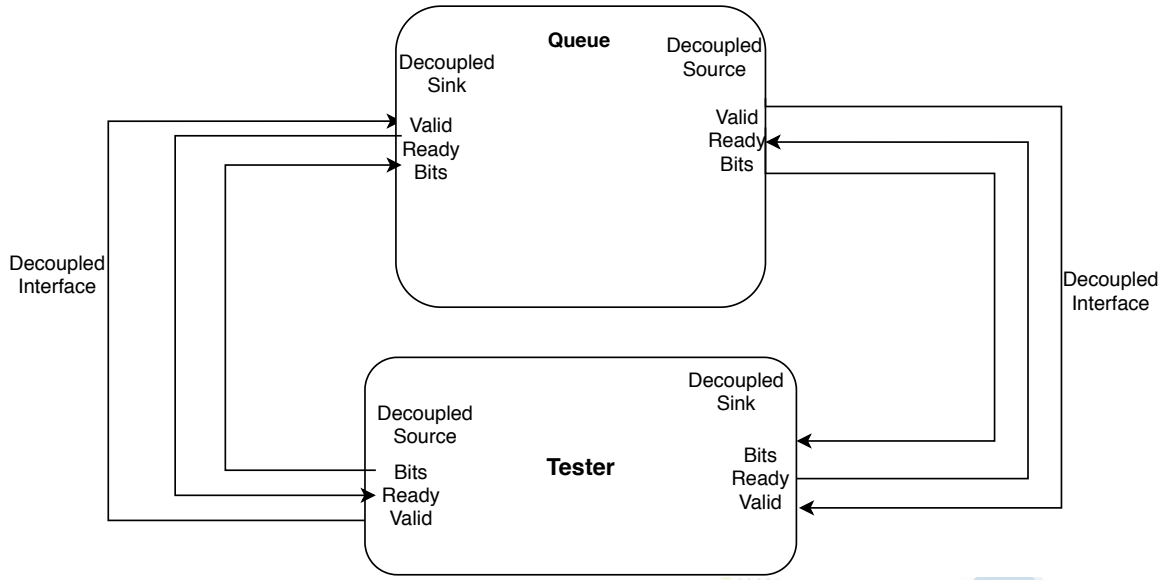


Figure 6.7: Queue with decoupled interface.

```
    val io  = IO(new Bundle {
        val in  = Flipped(Decoupled(UInt(8.W))) //valid = Input,  ready =
            Output, bits = Input
        val out = Decoupled(UInt(8.W))          //valid = Output, ready =
            Input , bits = Output
    })
    val queue = Queue(io.in, 5)                 // 5-element queue
    io.out <> queue
}
```

Listing 6.9: Queue with decoupled interface.

6.6 Black Boxes

Integrating existing Verilog IP is an essential part of many chip designs. Chisel provides support to integrate Verilog IP using `BlackBox`. The module defined as BlackBox will be instantiated in the generated Verilog, but no Verilog code will be generated to define the behavior of the module. In addition, there is no implicit clock or reset in BlackBox, which is similar to RawModule. So clock and reset needs to be explicitly declared and connected to BlackBox.

Let us use BlackBox for the implementation of a 32-bit Adder. Listing 6.10 shows the use of BlackBox to define the Adder module in Verilog. The Verilog source file is placed at the path src/main/resouces/Adder.v. Alternatively, we can also include Verilog code inline in the BlackBox as illustrated in Listing 6.11.

```
class BlackBoxAdder extends BlackBox with HasBlackBoxResource {
    val io = IO(new Bundle() {
        val in1 = Input(UInt(32.W))
        val in2 = Input(UInt(32.W))
        val out = Output(UInt(33.W))
    })
```

```
    setResource("/Adder.v")
}
```

Listing 6.10: Adder module using BlackBox.

```
class BlackBoxAdder extends BlackBox with HasBlackBoxInline {
    val io = IO(new Bundle() {
        val in1 = Input(UInt(32.W))
        val in2 = Input(UInt(32.W))
        val out = Output(UInt(33.W))
    })
    setInline("BlackBoxAdder.v",
    s"""
    |module BlackBoxAdder(
    | input [32:0] in1,
    | input [32:0] in2,
    | output [33:0] out
    |);
    |always @* begin
    | out <= ((in1) + (in2));
    |end
    |endmodule
    """.stripMargin)
}
```

Listing 6.11: Adder module using BlackBox with inline Verilog.

Since Chisel does not accept BlackBoxes as a top Module, to use BlackBox based Adder we need to instantiate in another module as given below.

```
val BBAdder = Module(new BlackBoxAdder)
```

6.7 Exercises

Exercise 1: Add parallel load capability to shift register implemented in Listing 6.2.

Exercise 2: Modify the counter implementation in the Listing 6.3 using the recommended modification from Listing 6.4.

Exercise 3: Modify one-shot timer in Listing 6.5 to work as a two-shot timer.

Exercise 4: Figure 6.8 contains three decoupled interfaces, each having a ready, valid and bits. Whenever both ready and valid are high it means its a valid transaction otherwise no transaction will occur. There are two queues, each has depth of '5' and width of UInt 8-bits. It can be seen from Figure 6.8 that decoupled interface between the queues is controlled by the queues automatically, while enqueue and dequeue operations are performed. However you can control the other two decoupled interfaces from testbench. Write Chisel code and simulate behavior using the skeleton code provided in Listing 6.12. Use decoupled interfaces and Queue constructs and explain the behavior of the module. Use io.in (decoupled interface) and io.out (decoupled interface) in your implementation.

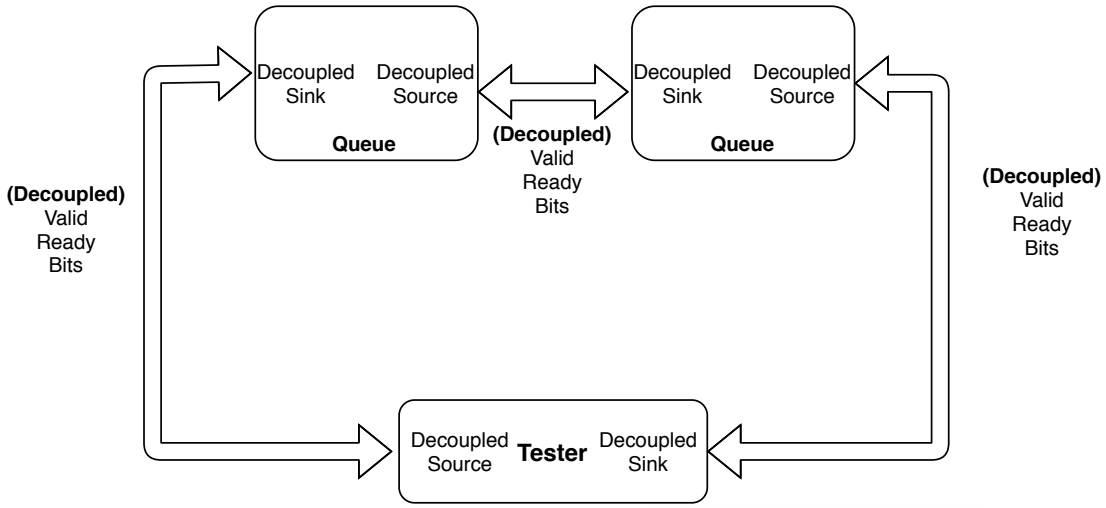


Figure 6.8: Decoupled and Queues.

```
package Lab6
import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec,Driver,PeekPokeTester}

class My_Queue extends Module{
// your code begin

// your code end here
}
```

Listing 6.12: Skeleton code for Problem 1.

Experiment 7

Finite State Machines

Objective

Learn to implement complex sequential circuits involving multiple states with arbitrary state transitions in Chisel.

7.1 Finite State Machine

No sequential circuit discussion is complete without finite state machines (FSMs). Finite state machines are constructed using the following three building blocks.

- Next state logic
- State register
- Output logic

At the start, the *state register* is initialized with the default state. The *next state logic* can be implemented by using either 'switch' or 'when' blocks or both. When using both, a possible scenario is where switch case will have 'when' block nested within it. The switch cases will determine the current state, while the 'when' blocks within it will be responsible for the change of state based on the input and current state. Finally, the *output logic* will be responsible to update the output.

7.1.1 FSM Types

The output logic implementation determines the type of finite state machine. In general, FSMs can be one of the following two types.

- **Mealy state machines:** Output depends on both state and input (see Figure 7.1(a) for illustration).
- **Moore state machines:** Output depends on state only (see Figure 7.1(b) for illustration).

7.1.2 An Example FSM

Figure 7.2 shows an FSM that can be used to detect a sequence ('110' in this case) of bits in a bitstream. This state machine has been implemented in Listing 7.1. Type enumeration is used to list and define the names for different state. The 'switch' and 'when' constructs are used to implement the state transition, as discussed previously.

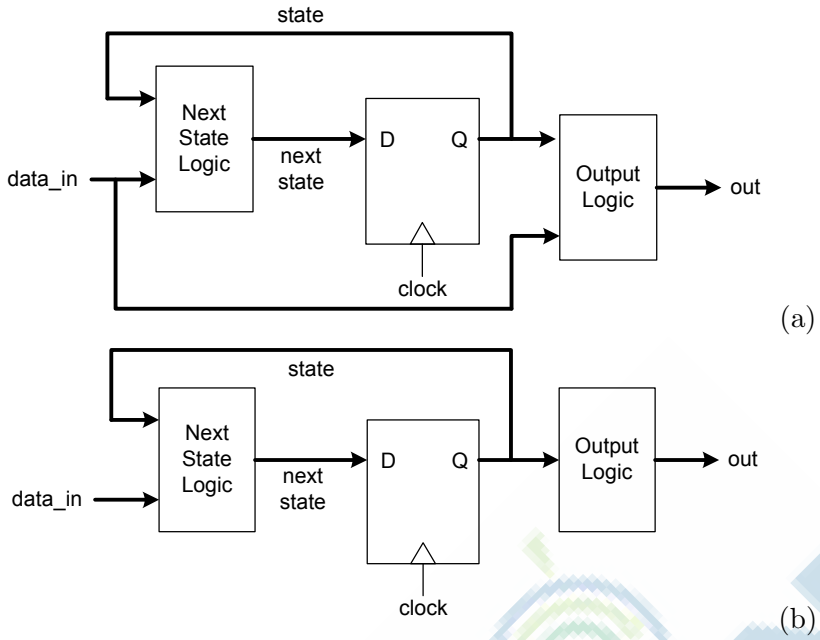


Figure 7.1: (a) Mealy type state machine, and (b) Moore type state machine.

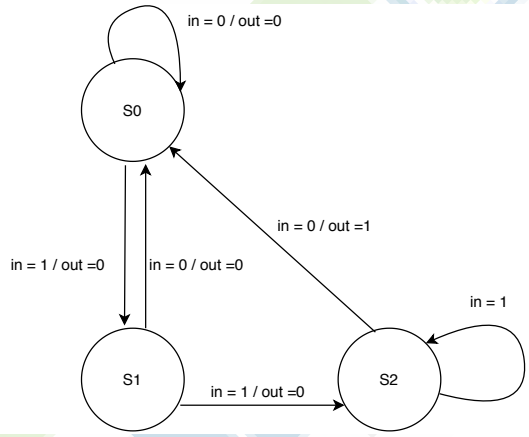


Figure 7.2: Sequence detector using an FSM.

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{
   ChiselFlatSpec, Driver, PeekPokeTester
}
import chisel3.experimental.ChiselEnum

//Sequence to detect is  110
class Detect_Seq extends Module {
   val io = IO(new Bundle {
      val in = Input(Bool())
      val out = Output(Bool())
   })

   val s0 :: s1 :: s2 :: Nil = Enum(3)    //Enumeration type
   val state = RegInit(s0)                //state = s0
```

```
    io.out := (state === s2) & (!io.in)    //Mealy type state machine

    switch (state) {
        is (s0) {
            when (io.in) {
                //move to next state when input is  1
                state := s1
            }
        }
        is (s1) {
            when (io.in) {
                //move to next state when input is  1
                state := s2
            } .otherwise {
                state := s0
            }
        }
        is (s2) {
            when (!io.in) {
                //move to default state when input is zero
                // otherwise stay here because input sequence is 111
                state := s0
            }
        }
    }
}
println(chisel3.Driver.emitVerilog(new Detect_Seq))
```

Listing 7.1: FSM for sequence detection.

7.2 Example: Up-down Counter

Next we discuss the implementation of an up-down counter using state machine. This counter is different from the one discussed previously in that it effectively generates a triangular waveform (when its count is plotted against time). The state transition diagram for the counter is shown in Figure 7.3. We observe that, once started, the counter keeps on toggling between up and down states. Listing 7.2 provides a possible implementation of the up-down counter and its test is given in Listing 7.3. The waveforms for different signals are shown in Figure 7.4.

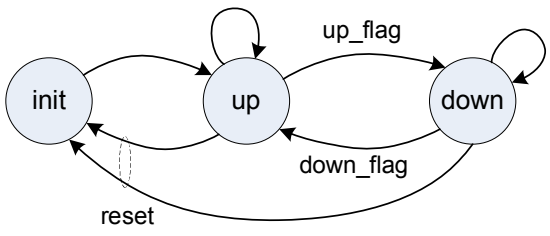


Figure 7.3: State transition diagram for up-down counter.

```
// up-down counter implementation

package LM_Chisel

import chisel3._
import chisel3.util._

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

    val counter = RegInit(0.U(n.W))
    val max_count = RegInit(6.U(n.W))

    val init :: up :: down :: Nil = Enum(3)    // Enumeration type
    val state = RegInit(init)                  // state = init
    val up_flag = (counter === max_count)
    val down_flag = (counter === 0.U)

    switch (state) {
        is (init) {
            state := up                         // on reset
        }

        is (up) {
            when (up_flag) {
                state := down
                // start count down immediately on up_flag
                counter := counter - 1.U
            }.otherwise {
                counter := counter + 1.U
            }
        }

        is (down) {
            when (down_flag) {
                state := up
                counter := counter + 1.U
                max_count := io.data_in     // load the counter
            }.otherwise {
                counter := counter - 1.U
            }
        }
    }
    io.out := up_flag | down_flag
}
object CounterUpDown_generate extends App {
    chisel3.Driver.execute(args, () => new CounterUpDown(8))
}
```

Listing 7.2: Up-down counter implementation.

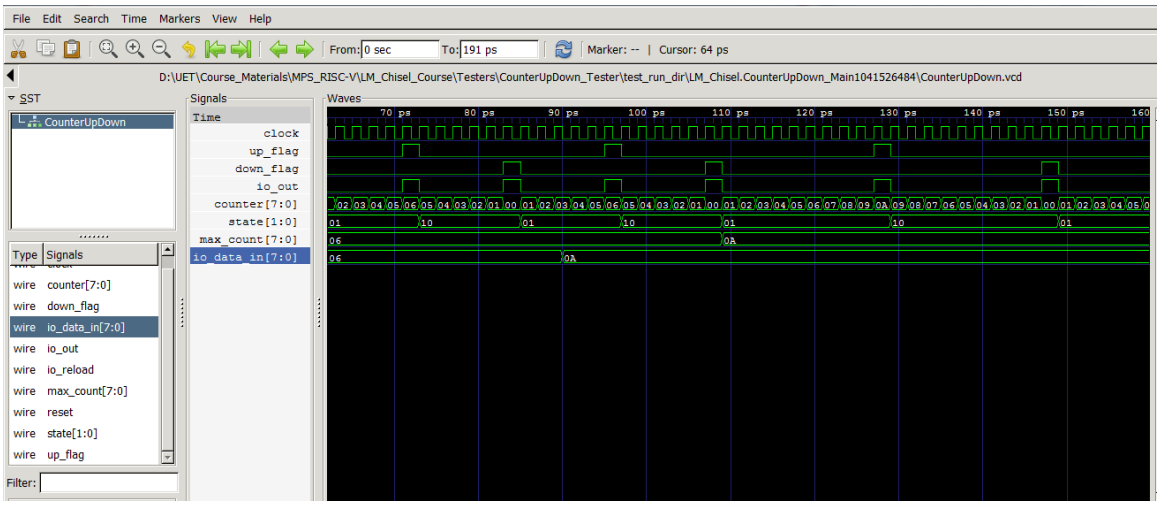


Figure 7.4: Up-down counter signal waveforms.

```
package LM_Chisel
import chisel3._
import chisel3.iotesters.{Driver, PeekPokeTester}

class TestCounterUpDown(c: CounterUpDown) extends PeekPokeTester(c) {
   var data_in = 6
   poke(c.io.data_in, data_in.U)
   step(40)
   data_in = 10
   poke(c.io.data_in, data_in.U)
   step(50)
}

// object for tester class
object CounterUpDown_Main extends App {
    iotesters.Driver.execute(Array("--is-verbose",
    "--generate-vcd-output","on", "--backend-name", "firrtl"),
    () => new CounterUpDown(8)) {c => new TestCounterUpDown(c)}
}
```

Listing 7.3: Up-down counter test.

7.3 Example: UART Transmitter

Finally we discuss the implementation of UART transmitter module, which involves, apart from state machine, many different Chisel constructs that we have discussed so far. The UART transmitter is responsible for serial data transmission at a predefined baudrate (bit rate). The baud rate frequency is derived from the clock, using the baud divisor parameter provided by the user, as part of UART transmitter implementation. Baud divisor along with other configuration parameters are passed by defining a parameter class as can be observed from the Listing 7.4. The implementation of the UART transmitter is provided in Listing 7.5.

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

```
import chisel3.util._

case class UART_Params(
    dataBits:    Int = 8,
    stopBits:    Int = 2,
    divisorBits: Int = 5,
    oversample:  Int = 2,
    nSamples:    Int = 3,
    nTxEntries:  Int = 4,
    nRxEntries:  Int = 4) {
  def oversampleFactor = 1 << oversample
  require(divisorBits > oversample)
  require(oversampleFactor > nSamples)
}
```

Listing 7.4: UART configuration parameters.

```
class UART_Tx(c: UART_Params) extends Module {
  val io = IO(new Bundle {
    val en    = Input(Bool())
    val in    = Flipped(Decoupled(UInt((c.dataBits).W)))
    val out   = Output(Bool())
    val div   = Input(UInt((c.divisorBits).W))
    val nstop = Input(UInt((c.stopBits).W))
  })
  // pulses generated at baud rate using prescaler
  val prescaler = RegInit(0.U((c.divisorBits).W))
  val pulse     = (prescaler === 0.U)
  private val n = c.dataBits + 1

  val counter = RegInit(0.U((log2Floor(n + c.stopBits)+1).W))
  val shifter = Reg(UInt(n.W))
  val out     = RegInit(true.B)
  io.out      := out

  val busy    = (counter =/= 0.U)
  val state1  = io.en && !busy
  val state2  = busy
  io.in.ready := state1

  when(state1) {
    shifter := Cat(io.in.bits, false.B)
    counter := Mux1H(
      (0 until c.stopBits).map(i => (io.nstop === i.U) -> (n+i+2).U)
    )
  }

  when(state2) {
    prescaler := Mux(pulse, (io.div - 1.U), prescaler - 1.U)

    when(pulse) {
      counter := counter - (1.U)
      shifter := Cat(true.B, shifter >> 1)
```

```
      out := shifter(0)
    }
  }
}

// Instantiation of the UART_Tx module for Verilog generator
object UART_Tx_generate extends App {
val param = UART_Params()
  chisel3.Driver.execute(args, () => new UART_Tx(param))
}
```

Listing 7.5: UART transmitter implementation.

7.4 Arbiter

An arbiter is a device that follows producer/consumer model and is responsible to sequence n producers to one consumer. There are different types of arbiters depending on the policies they implement. The Chisel library provides two types of arbiters.

7.4.1 Priority Arbiter

The priority arbiter in Chisel utilities library is referred as *Arbiter*. When ever we make an arbiter we need to specify the number of requestors and the data type with width of the data coming from requestors. Each requestor should be Decoupled and Flipped before connecting to the arbiter. The output of the arbiter is also decoupled. A priority arbiter supporting 2 requestors is implemented in Listing 7.6 using `Arbiter` construct.

```
val arb_priority = Module(new Arbiter(UInt(), 3))

// connect the inputs to different producers
arb_priority.io.in(0) <> producer0.io.out
arb_priority.io.in(1) <> producer1.io.out
arb_priority.io.in(2) <> producer2.io.out

// connect the output to consumer
consumer.io.in <> arb_priority.io.out
```

Listing 7.6: Priority arbiter using `Arbiter` construct.

7.4.2 Round Robin Arbiter

Round robin arbiter is an arbiter that implements round robin policy for arbitration. A round robin arbiter can be implemented using the `RRArbiter` construct as illustrated in Listing 7.7.

```
val arb_noPriority = Module(new RRArbiter(UInt(), 3))

// connect the inputs to different producers
arb_noPriority.io.in(0) <> producer0.io.out
arb_noPriority.io.in(1) <> producer1.io.out
```

```
arb_noPriority.io.in(2) <> producer2.io.out

// connect the output to consumer
consumer.io.in <> arb_noPriority.io.out
```

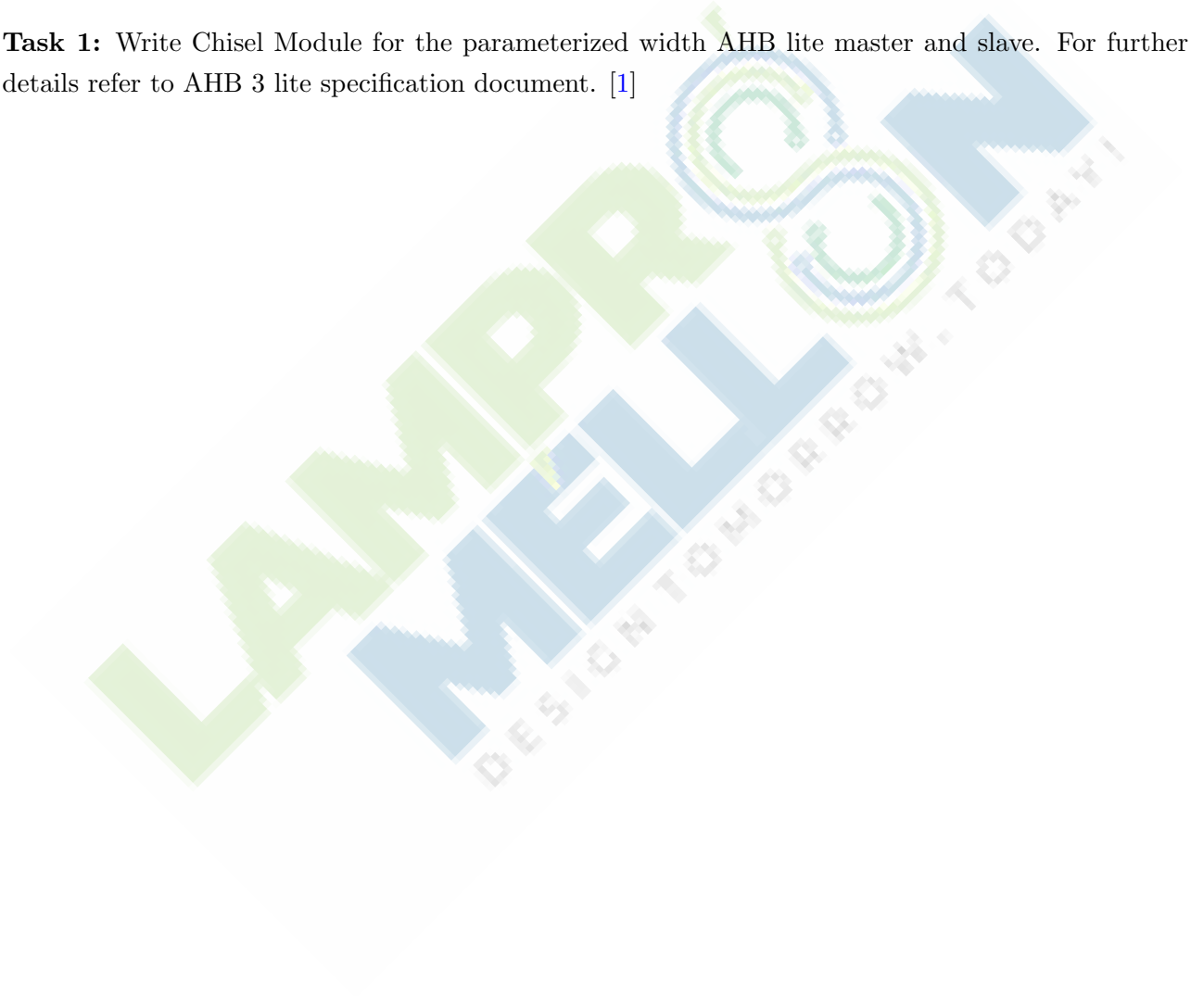
Listing 7.7: Round robin arbiter using `RRArbitor` construct.

7.5 Exercises

Exercise 1: Connect `Arbiter` with two `Queues` at the input and perform testing using `PeekPokeTester`.

7.6 Assignments

Task 1: Write Chisel Module for the parameterized width AHB lite master and slave. For further details refer to AHB 3 lite specification document. [1]



Experiment 8

Memories

Objective

Understand different memory constructs available in Chisel and learn their use for instruction and data storage.

8.1 Memory in Chisel

Memories can be synchronous as well as asynchronous. In addition, the memories can have multiple ports. In Chisel, we can define the following two types of memories with single port memory interface.

- Asynchronous read, synchronous write
- Synchronous read and synchronous write

8.1.1 Asynchronous Memory

Asynchronous memories can be made in *Chisel* using `Mem` construct. We have already seen asynchronous memory in Lab 2. There are two methods associated with `Mem`.

- **Write:** To perform write operation we call method *write* on the `Mem` object and provide data and address address as arguments. The data is written to the specified address on the positive edge of the clock.
- **Read:** The Read method defined on `Mem` object is combinational, which implies data is available immediately after applying the address. Depending on the memory model used for `Mem` implementation, immediately, can imply either no delay or some constant delay corresponding to combinational implementation of address decoding with some propagation delay offset. When performing a read operation, we only need to provide address as an argument.

The implementation of asynchronous memory using `Mem` and use of read and write methods is illustrated in Listing 8.1

```
// mem Example
import chisel3._

class IO_Interface extends Bundle{
   // Make an input from a Vector of 4 values
   val data_in = Input(Vec(4,(UInt(32.W))))

   // Signal to control which vector is selected
```

```
    val data_selector = Input(UInt(2.W))
    val data_out = Output(UInt(32.W))
    val addr = Input(UInt(5.W))

    // The signal is high for write
    val wr_en = Input(Bool())
}

class Asynch_Mem extends Module {
    val io = IO(new IO_Interface)

    io.data_out := 0.U

    // Make a memory of 32x32
    val memory = Mem(32, UInt(32.W))

    when(io.wr_en){
        // Write for wr_en = 1
        // Write at memory location addr, with selected data from data_in
        memory.write(io.addr, io.data_in(io.data_selector))
    }
    // Asyncronous read from addr location
    io.data_out := memory.read(io.addr)
}

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

Listing 8.1: Asynchronous memory using `Mem`.

8.1.2 Synchronous

Synchronous memories have read operation also synchronized to the clock. We can make synchronous memories in *Chisel* using `SyncReadMem`. When instantiating memory using `SyncReadMem`, we need to specify memory size in terms of number of locations as well as data-type and width of each location. For instance, we can implement word addressable synchronous memory with 1024 32-bit locations, using the following syntax.

```
val memory = SyncReadMem (1024, UInt(32.W))
```

8.1.3 Memory Parameterization

Memory generation can be parameterized by specifying memory size as well as the size of the smallest addressable location as parameters. Listing 8.2 provides an illustration to implement parameterized memory.

```
// parameterized memory
import chisel3._
import chisel3.util._
```

```
class Parameterized_Mem(val size: Int = 32, val width: Int = 32) extends
    Module {
    val io = IO(new Bundle {
        val dataIn = Input(UInt(width.W))
        val dataOut = Output(UInt(width.W))
        val addr = Input(UInt(log2Ceil(size).W))
        val rd_enable = Input(Bool())
        val wr_enable = Input(Bool())
    })

    val Sync_memory = SyncReadMem(size, UInt(width.W))
    // memory write operation
    when(io.wr_enable){
        Sync_memory.write(io.addr, io.dataIn)
    }
    io.dataOut := Sync_memory.read(io.addr, io.rd_enable)
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Parameterized_Mem))
```

Listing 8.2: Parameterized memory using SyncReadMem.

8.1.4 Implementing Register File

We have implemented a register file using Vec in Lab 6. We can also implement the register file using memory. This can be easily achieved by replacing the register vector with memory as illustrated in the following syntax.

```
val regFile = Mem (32, UInt(32.W))
```

8.1.5 Initializing Code memory

Code memory can be implemented using either synchronous or asynchronous memories. When testing the functionality of a processor, one needs to load the code memory with the executable of user program. When simulation is the mode of testing, then one needs to initialize the code memory with the user program. This can be achieved by using loadMemoryFromFile from Chisel library utility. Listing 8.3 illustrates this functionality.

```
package LM_Chisel

import chisel3._
import chisel3.util._
import chisel3.util.experimental.loadMemoryFromFile
import scala.io.Source

class InstMemIO extends Bundle with Config {
    val addr = Input(UInt(WLEN.W))
    val inst = Output(UInt(WLEN.W))
```

```
}

class InstMem(initFile: String) extends Module with Config {
    val io = IO(new InstMemIO)

    // INST_MEM_LEN in Bytes or INST_MEM_LEN / 4 in words
    val imem = Mem(INST_MEM_LEN, UInt(WLEN.W))

    loadMemoryFromFile(imem , initFile)

    io.inst := imem (io.addr / 4.U)
}
```

Listing 8.3: Initializing code memrory.

User executable file, to be loaded to instruction memory, is passed as string parameter by the top module as illustrated in Listing 8.4. A separate .v file is generated during hardware generation that is used for binding instruction memory to executable file.

```
object Generate_ProcessorTile extends App {
    var initFile = "src/test/resources/main.txt"

    chisel3.Driver.execute(args, () => new ProcessorTile(initFile))
}
```

Listing 8.4: Top module passing user executable file as paramater.

8.1.6 Memory with Forwarding

Forwarding is required when write and read operations are performed on the same memory location during the same cycle. This can happen with memories supporting simultaneous read and write operations [2]. Contrary to this, memory with sequential read and write operations does not encounter this problem. Memory interfaces using sequential and simultaneous read-write operations are depicted in Figure 8.1.

In forwarding, the read operation returns the new write value instead of the previously stored value. To achieve compatibility with the expected one-cycle read-after-write latency behavior, the write address and data are put through register to delay by one cycle. Block diagram illustrating forwarding implementation is shown in Figure 8.2. The implementation of forwarding logic is given in Listing 8.5.

```
// Memory forwarding example
import chisel3._
import chisel3.util._

class Forwarding extends Module {
    val io = IO(new Bundle{
        val out = Output(UInt(32.W))
        val rdAddr = Input(UInt(10.W))
```

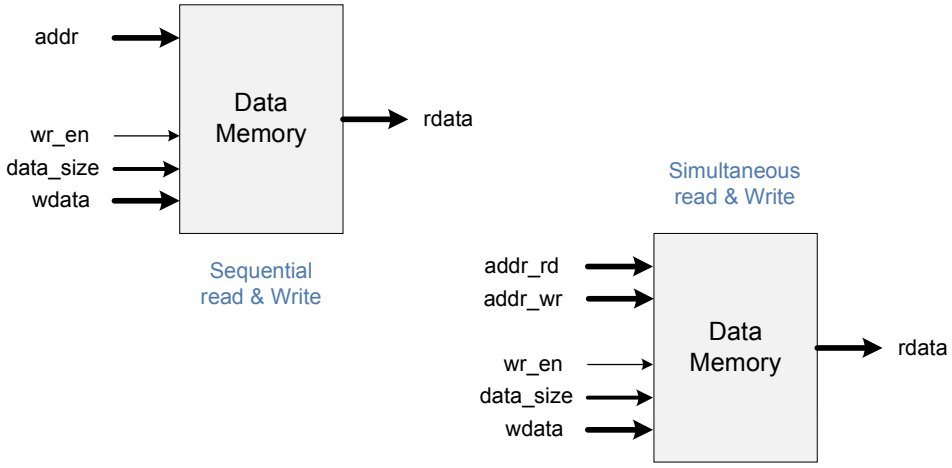


Figure 8.1: Data memory interfaces.

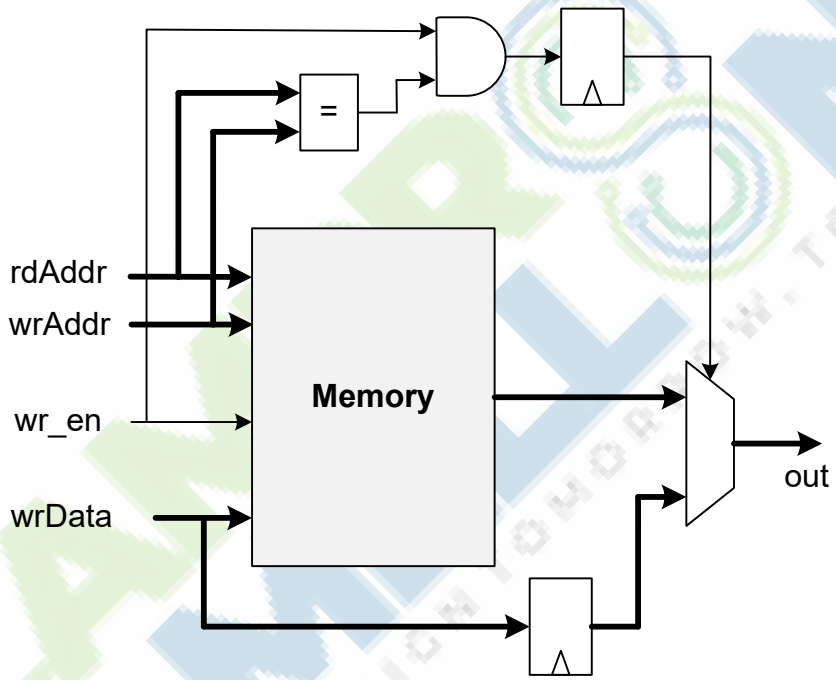


Figure 8.2: Memory forwarding.

```
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(32.W))
    val wr_en = Input(Bool())
})

val memory = SyncReadMem(1024, UInt(32.W))
val wrDataReg = RegNext(io.wrData)
val doForwardReg = RegNext(io.wrAddr === io.rdAddr && io.wr_en)
val memData = memory.read(io.rdAddr)
when(io.wr_en)
{
    memory.write(io.wrAddr, io.wrData)
}
io.out := Mux(doForwardReg , wrDataReg , memData)
```

```
}

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

Listing 8.5: Memory forwarding using mem.

8.1.7 Memory with Mask

Previously, we have used write method with SyncReadMem using two arguments, that is address and data. However, the write method also supports a third argument, which is the *mask*. Masking is required when performing byte and half-word access from a word addressable location. To illustrate the use of mask we define memory with 32-bit width and can be byte masked.Implementation of such a memory is illustrated in Listing 8.6.

```
import chisel3._
class MaskedReadWriteSmem extends Module {
   val width: Int = 8
   val io = IO(new Bundle {
      val enable = Input(Bool())
      val write = Input(Bool())
      val addr = Input(UInt(10.W))
      val mask = Input(Vec(4, Bool()))
      val dataIn = Input(Vec(4, UInt(width.W)))
      val dataOut = Output(Vec(4, UInt(width.W)))
   })

   // Create a 32-bit wide memory that is byte-masked
   val mem = SyncReadMem(1024, Vec(4, UInt(width.W)))
   // Write with mask
   mem.write(io.addr, io.dataIn, io.mask)
   io.dataOut := mem.read(io.addr, io.enable)
}

println(chisel3.Driver.emitVerilog(new MaskedReadWriteSmem))
```

Listing 8.6: Memory with mask.

8.2 Exercises

Exercise 1: Modify Listing 8.6 without using overloaded method for masking.

Exercise 2: Implement forwarding for a dual-bank memory. Refer to Listing 8.6.

Exercise 3: Write instruction stream to a memory and verify it.

8.3 Assignments

Task 1: Write Chisel code for parametrized cache complained with AHB lite. Parametrized with number of sets, number of ways and size.

Experiment 9

Scala Collections

Objective

To familiarize with important Scala collections and supported methods for efficient hardware construction.

9.1 Scala Collections

Scala has rich collections libraries. Collections serve as containers for data of similar or different types, which can be efficiently operated by the supported methods of the respective collection library. Scala collections are grouped in three packages or sub-packages. Using the keyword `Seq` implies immutable collection, while `mutable.Seq` will refer to mutable counterpart. Figure 10.1[1] shows the collections hierarchy for immutable collections.

- `scala.collection`: Includes both mutable and immutable collections
- `scala.collection.mutable`: Includes mutable collections
- `scala.collection.immutable`: Includes immutable collections

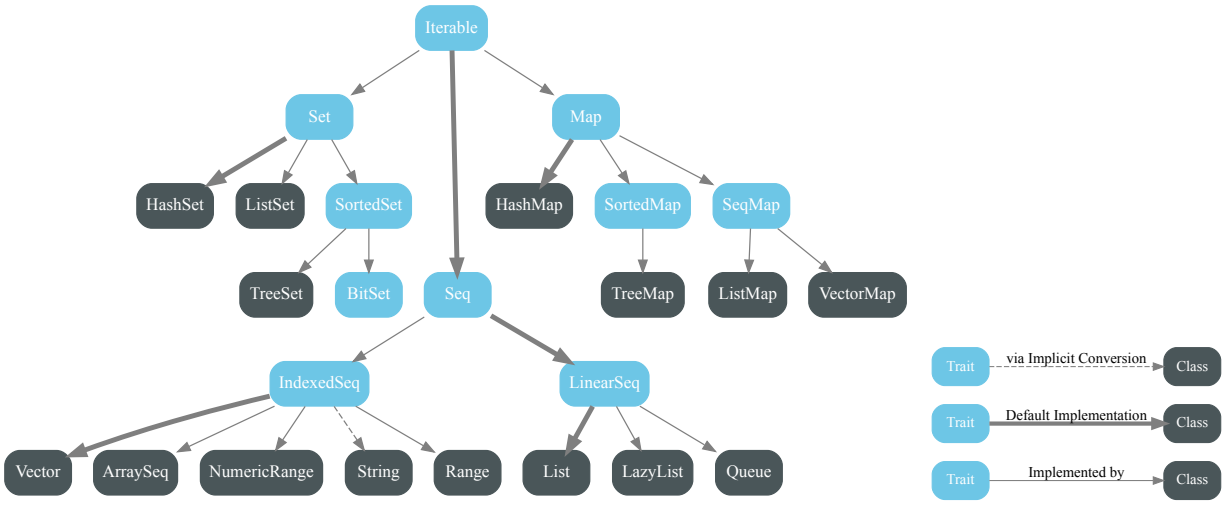


Figure 9.1: Immutable collections tree.

9.1.1 Arrays

Arrays are mutable collection of the same data type. The syntax to define an Array is given in Listing 9.1.

[1]Retrieved from: https://docs.scala-lang.org/overviews/collections-2.13/overview.html

```
val arr1 : Array [Int] = new Array [Int](4)
// Two key things to see here are
// 1 - We are explicitly telling the data type of to be Array of Int
// 2 - We are telling the length of the Array to be 4
// We can do both of these things to be implicitly inferred

// Inferring data type
val arr2 = new Array [Int](5)

// Inferring data type and width
val arr3 = Array (1, 2, 3, 4, 5, 6)
```

Listing 9.1: Declaring Arrays

9.1.2 List

At first look, Lists are similar to Arrays. Like Array, in a List, all the elements must have the same data type. But there are two key differences between Arrays and Lists.

- Lists are immuatble whereas arrays are mutable.
- Lists are stored as linked lists, whereas arrays are stored as a single block of memory.

The syntax for declaring a list is shown below.

```
val lst: List[Int] = List(1, 2, 3, 4, 5)
```

Listing 9.2: Declaring a List

We can append and prepend to a list by using :: operator. We can even make a complete List using :: operator. Similarly, we can concatenate two lists using `:::` or `List.:::()` or `List.concat()`.

```
val lst = (1::2::3::4::5::Nil)
val modules: List[String] = List("ALU", "Branch", "Control")
val peripherals = List("Uart", "Timer")

// concatenate two lists
val combined_list = modules ::: peripherals
println(combined_list)  // List(ALU, Branch, Control, Uart, Timer)
```

Listing 9.3: Making a List using `::` operator

There are many methods that can be used on Lists and many of them can be used with other collections as well. For instance, we have methods `.head`, `.tail`, `.size`, `.length`, to name few, that can operate on a List (see Listing 9.4 for reference.). Explore other methods that are available with List and Array collections.

```
val lst =  (1::2::3::4::5:: Nil)
val first_element = lst.head //returns first element
```

```
println( s"First element of lst is = ${
    first_element
}")
```

Listing 9.4: Using head method.

9.1.3 Set

Sets have the base trait that there cannot be duplicate elements in a Set. At the time of declaration if a set has an element with duplicate entries, then all the duplicate elements will be removed. All elements in a Set must have same data type. Key methods that are available with Set are.head, .tail, .isEmpty, .foreach, .++ and .&. The method .++ can be used to get the union of two sets. Whereas .& can be used to get the intersection of two sets. Listing 9.5 illustrates the use of sets. We can do much more with Sets other than taking union and intersection as Scala provides many useful methods to manipulate sets.

```
//The i in "iSet1" shows that it is immutable
val iSet1 : Set[Int] = Set(1, 2, 3, 4, 5)
val iSet2 : Set[Int] = Set(4, 5, 6, 7, 8)
println(s"Union of both Sets are =${
    iSet1.++(iSet2)
}")
// Union of the two Sets = Set(5, 1, 6, 2, 7, 3, 8, 4)
println(s"Intersection of both Sets are =${
    iSet1.&(iSet2)
}")
// Intersection of the two Sets = Set(5, 4)
```

Listing 9.5: Union and Intersection of Sets.

9.1.4 Map

The Map collection works as a key-value pair. Keys in a Map are always unique, so keys are actually a set, in which a value is associated with each member of the set. Values can be duplicate. Keys and values can have different data types.

```
val Capitals:Map[String,String] = Map("Pakistan" -> "Islamabad", "China" ->"
    Beijing", "Japan" -> "Tokoyo")
println ( s"The capital of Pakistan is ${
    Capitals("Pakistan")
}.")
//The capital of Pakistan is Islamabad.
```

Listing 9.6: Example of Map Collection

Both Set and Map collections have mutable and immutable flavors. Here we are using a default type of Map, which is immutable.

9.1.5 Tuple

If we want to have different data types within a collection we need to make it a tuple. Tuples can have a maximum of 22 elements. However, each element can itself be a tuple. This is termed as tuple of a tuple. Let's make a tuple and try to print out its element. We define a collection that has an Int, String, and a Bool type data elements. The method to access the elements of the tuple is different from other collections. This is illustrated by accessing the second last element using `tup._2`. Next example illustrates a tuple of the tuple.

```
val tup = (1 , "LM", false )
println ( s" Data at location 2 is : ${
    tup._2
}")
// Data at location 2 is : LM
```

Listing 9.7: Declaring a Tuple

```
val tup_of_tup = (1 , "LM", false , (1 , 2 , 4 , " Lab#9") )
println ( s" This is an example from${
    tup_of_tup._4._4
}")
```

Listing 9.8: Tuple of a tuple

9.1.6 Options

Sometime we are not sure if a collection contains a certain element or not. In such cases we can use option type. Option type in Scala is referred to as a carrier having one or no element of the stated type. We can check the presence of a specific element using the `get` method. For instance, the get method for the collection returns `Some` if the value is found, otherwise it returns `None`. Options can also be used in other situations. For example, it can be used as a function parameter.

```
val mapCollection = Map (2 ->"1st", 4 -> "2nd", 8 ->"3rd", 16 ->"4th", 32 ->
    "5th", 64 ->"6th")
println(mapCollection.get(2))
//Some(1st)
println(mapCollection.get(3))
//None
println(mapCollection.get(3).getOrElse("Not in the collection"))
//Not in the collection
```

Listing 9.9: Using `get` method on a Map

9.1.7 Iterator

Iterator is a collection that works similar to a queue. We can get the head by calling `next` method on it. To check if the iterator is empty we call `hasnext` method on it. Method `hasnext` returns a Boolean value to tell if iterator is empty or not.

You can only go through an Iterator once because it is consumed as you go through it. The reason for using an Iterator is generally for performance and memory benefits. There are methods that return an Iterator, for example, the .combinations method returns an Iterator.

```
val iterate = Iterator ("Scala","is","fun")
while (iterate.hasNext)
println (iterate.next())
//Scala is fun
```

Listing 9.10: Declaring an Iterator

9.2 Controller Design

In this section we will look into the controller implementation for a RISC V microprocessor, which takes advantage of different Scala collections. Controller implementation is one of the most critical steps in the processor realization. In Listing 9.11 different control signals, including inputs to the controller and outputs from the controller, have been defined as a concrete class.

```
class ControlSignals extends Bundle with Config {
    val inst       = Input(UInt(XLEN.W))
    val pc_sel     = Output(UInt(2.W))
    val inst_kill  = Output(Bool())
    val A_sel      = Output(UInt(1.W))
    val B_sel      = Output(UInt(1.W))
    val imm_sel    = Output(UInt(3.W))
    val alu_op     = Output(UInt(5.W))
    val br_type    = Output(UInt(3.W))
    val st_type    = Output(UInt(2.W))
    val ld_type    = Output(UInt(3.W))
    val wb_sel     = Output(UInt(2.W))
    val wb_en      = Output(Bool())
    val csr_cmd    = Output(UInt(3.W))
    val illegal    = Output(Bool())
    val en_rs1     = Output(Bool())
    val en_rs2     = Output(Bool())
}
```

Listing 9.11: Control signals definitions.

The input to the controller is a RISC V instruction, which is looked up from the control table and corresponding control signals are generated. The control signals are used by different modules of the processor to perform their function in the context of respective instruction. The control table with partial entries is provided in Listing 9.12[2]. In Listing 9.13, a Controller is implemented using `ListLookup`, which is a Chisel construct.

```
object Control {
```

[2]Derived from the controller of riscv-mini (https://github.com/ucb-bar/riscv-mini)

```
val default =
//                                                                  kill                                wb_en  illegal?  en_rs2
//            pc_sel  A_sel   B_sel  imm_sel   alu_op   br_type |  st_type ld_type wb_sel  | csr_cmd | en_rs1 |
//              |       |       |      |          |         |    |      |        |        |  |       |  |     |
List(PC_4,   A_XXX,  B_XXX, IMM_X, ALU_XXX   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, Y, N,   N)
val map = Array(
LUI   -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_COPY_B, BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,   N),
AUIPC -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_ADD   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,   N),
JAL   -> List(PC_ALU, A_PC,   B_IMM, IMM_J, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, N,   N),
JALR  -> List(PC_ALU, A_RS1,  B_IMM, IMM_I, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, Y,   N),

BEQ   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_EQ , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BNE   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_NE , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BLT   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_LT , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BGE   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_GE , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BLTU  -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_LTU, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BGEU  -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_GEU, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
```

Listing 9.12: Control signals for different instructions.

```
class Control extends Module {
   val io = IO(new ControlSignals)
   val ctrlSignals = ListLookup(io.inst, Control.default, Control.map)

   // Control signals for Fetch
   io.pc_sel    := ctrlSignals(0)
   io.inst_kill := ctrlSignals(6).toBool

   // Control signals for Execute
   io.A_sel   := ctrlSignals(1)
   io.B_sel   := ctrlSignals(2)
   io.imm_sel := ctrlSignals(3)
   io.alu_op  := ctrlSignals(4)

   ...
```

Listing 9.13: Controller implementation using ListLookup.

9.3 Exercises

Exercise 1: Set and Map collections can also be mutable. Figure out how to make them mutable.

Exercise 2: Find out how to use Option type as a function parameter.

Exercise 3: In this session we saw some methods like `.head` and `.tail` that can be applied on Lists and Arrays. Some of these methods are called higher order methods and we can pass a function as parameter to these methods. Explore the higher order methods available in Scala for arrays. This will be helpful in doing task 4.

9.4 Assignments

Task 1: Generate a list of 15 integer numbers randomly from 50 - 500. After making the complete list check if each element is prime or not, if its a prime number then put it into an iterator. Finally sort

them in ascending order and put them into a Map. Where each key should be the element location of the number. Bonus point for anyone who does not use `for` loop.

Task 2: Write a function that returns a `List[Char]` containing 'a'-'z' using tail recursion. The only argument which is passed to the method is the start alphabet array in ASCII i.e 98. *Hint*: Use `toChar` to make this work.

```
def CharArray(start:Int):List[Char]={
    //code here
}
```

Task 3: Given two `Array[Double]` values of the same length, write a function that returns the element-wise sum. This is a new Array where each element is the sum of the values from the two input arrays at that location. So if you have Array(1,2,3) and Array(4,5,6), the result should be Array(5,7,9).

Task 4: Implement different solutions that will take an `Array[Int]` and return number of even values in the Array. To test this on a large array, you can make one using

```
Array.fill(100)(util.Random.nextInt(100))
```

1. Use a recursive function.
2. Use the count higher-order method.

Task 5: Implement the following function that will build a Map from any sequence of a type with a function that can make keys from values.

```
def buildMap [A,B]( data : Seq [A], f: A => B): Map [B,A]{
    // code here
}
```

where 'f' is a user defined function which is passed as parameter. Below is an example of how we can use buildMap method to make a Map collection.

```
// Example
val lst = Array (1 ,2 ,3 ,4 ,5)
def func (x: Int ): Boolean = x%2 == 0 // entry is even or not
val result = buildMap (lst , func )
// Output
// result : Map [Int , Boolean ] = Map (1 -> false , 2 -> true , 3 -> false
    , 4 ->
true )
```

Experiment 10

Scala I

Objective

Learn to use Scala wildcard and zip/unzip, reduce, fold methods with Scala collections with illustrations from Rocket chip.

10.1 Scala Wildcard '_'

A wildcard is a kind of placeholder represented by a single character, which happens to be '_' in Scala. This attribute is also present in other programming languages, e.g. in Java, we use '*' for wildcard. It will be illustrated in later sections that how a wildcard can also be used as a default case handler, to access a tuple selectively as well as to import classes or functions, to name a few. For instance, Listing 10.1 shows how a wildcard can be used to import necessary packages and libraries.

```
import scala.collection._
import chisel3._
```

Listing 10.1: Importing dependencies

10.1.1 Wildcard as Placeholder

Let us first see how a wildcard character can be used as a placeholder. In Listing 10.2, a list is instantiated with both positive and negative integers. It is then modified using the `filter` method to remove or filter all negative integers. This is first done by calling filter method with explicit arguments. Next we perform the same filtering by using _ wildcard as placeholder for the arguments to the filter method. The output for both approaches will be the same as can be noted from Listing 10.2.

```
// An example List
val uList = List(11, -10, 5, 0, -5, 10)

// Applying .filter to List
val uList_filter1 = uList.filter(x => x > -1)
println(s"Filtered list = $uList_filter1")

// Applying .filter to List using _
val uList_filter2 = uList.filter(_ > -1)
println(s"Filtered list using _ as place holder = $uList_filter2")

// The output at the terminal
Filtered list = List(11, 5, 0, 10)
```

```
Filtered list using _ as place holder = List(11, 5, 0, 10)
```

Listing 10.2: Wildcard in filter method.

Listing 10.3 shows another use of wildcard in a Scala `match` statement. Here, the user defined method ALU_Scala checks a list of possible cases against an input argument 'op' and then applies the respective operation on the input operands. Observe that for the last case, a wildcard character is used to define the default case. This is equivalent to the scenario, where the match function can not find a matching case against the 'op' argument supplied and as a result it selects the default case.

```
def ALU_Scala(a: Int, b: Int, op: Int): Int = {
    op match {
        case 1 => a + b
        case 2 => a - b
        case 3 => a & b
        case 4 => a | b
        case 5 => a ^ b
        case _ => -999    // This should not happen
    }
}

var result = ALU_Scala(18,11,2)
println(s"The result is: $result")

result = ALU_Scala(12,17,9)
println(s"The result is: $result")
```

Listing 10.3: Wildcard in default match case.

Similar to what we did with the filter method, we can define our own methods and use one or multiple wildcards, when calling these user defined methods. Listing 10.4 illustrates this through an example. The user defined function 'compose' defines two integer type functional mappings as well as the order of the mappings. Functions y1 and y2 are defined using the 'compose' function and provide the expressions for the mapping functions. Both y1 and y2 are the same except that y2 uses wildcard to write the function expressions. Both functions are called on the same list i.e 'uList' and produce the same output.

```
// User type definition
type R = Int

// An example List
val uList = List(1, 2, 3, 4, 5)

// functional composition
def compose(g: R => R, h: R => R) =
(x: R) => g(h(x))

// implement y = mx+c ( with m=2 and c =1)
def y1 = compose(x => x + 1, x => x * 2)
```

```
def y2 = compose(_ + 1, _ * 2)

val uList_map1 = uList.map(x => y1(x))
val uList_map2 = uList.map(y2(_))

println(s" Linearly mapped list 1 = $uList_map1 ")
println(s" Linearly mapped list 2 = $uList_map2 ")
```

Listing 10.4: Wildcard as a function argument.

10.1.2 Wildcard for Function Assignment

Another possible use of wildcard is to assign methods, called from libraries, to variables as illustrated in Listing 10.5. Two methods `max` and `abs` are assigned to two variables and then instead of calling these two methods using long chained names, we can simply use variables as function callers. The required arguments to make function calls, are supplied the same way as one would do with the original function call.

```
// assigning a function to val
val f_max = scala.math.max _
val f_abs = scala.math.abs _

// calling the function
val max_value = f_max(1, 5)
val abs_value = f_abs(-123)

println(s"The maximum value is = $max_value")
println(s"The absolute value is = $abs_value")
```

Listing 10.5: Wildcard for assigning functions.

10.1.3 Higher Order Functions

Higher order functions in Scala are those functions that either use other function(s) as argument or return a function. The function arguments are supplied along with other operands and the parent function may call these function(s) in the body. Such a function is given in Listing 10.6. The two instances of 'higherOrder' function call illustrate the function argument provided explicitly as well as using wildcard.

```
// Define a higher order function
def higherOrder(a: Int, b: Int, c: Int, d: Int, function: (Int, Int) => Int)
    = {
   val first_inst = function(a, b)
   val second_inst = function(c, d)
   function(first_inst, second_inst)
}

// call to higherOrder function
```

```
val result1 = higherOrder(2, 5, 7, 9, (x, y) => x + y)
val result2 = higherOrder(2, 5, 7, 9, _ + _)

println(s"The result1 is = $result1")
println(s"The result2 is same = $result2")
```

Listing 10.6: Higher order functions.

10.1.4 Hiding a Class During Import

When importing a library, a specific class or object can be prevented from being imported by using wildcard as illustrated in Listing 10.7. Listing 10.8 shows the error generated at the output terminal when we attempt to call the prevented class.

```
import chisel3.util.{MuxCase => _ , _}

io.out := MuxCase(false.B, Array(
    (io.sel === 0.U)  ->   io.in0,
    (io.sel === 1.U)  ->   io.in1,
    (io.sel === 2.U)  ->   io.in2,
    (io.sel === 3.U)  ->   io.in3 )
)
```

Listing 10.7: Leaving a class out.

```
// We get the following error message
not found: value MuxCase
```

Listing 10.8: Error while calling class.

10.2 The apply method

The `apply` method is widely used as one of the default methods supported by Scala traits, classes and objects. For the three traits Seq, Set and Map the `apply` method functioning is summarized below.

- Seq: `apply` is positional indexing, element numbering starts from 0 always
- Set: `apply` is a membership test and is similar to the contains method
- Map: `apply` is a selection and returns the value associated with the given key

Based on the above discussion, the use of `apply` method is illustrated in Listing 10.9 for three different collections, namely, List, Set and Map. Figure 10.1, showing the collections hierarchy for immutable collections, is reproduced here for quick reference.

```
// Illustration of built-in apply functions for different types of collections
val uList = List (11 , 22 , 33 , 44 , 55)
val uSet = Set (11 , 22 , 33 , 44 , 55)
```

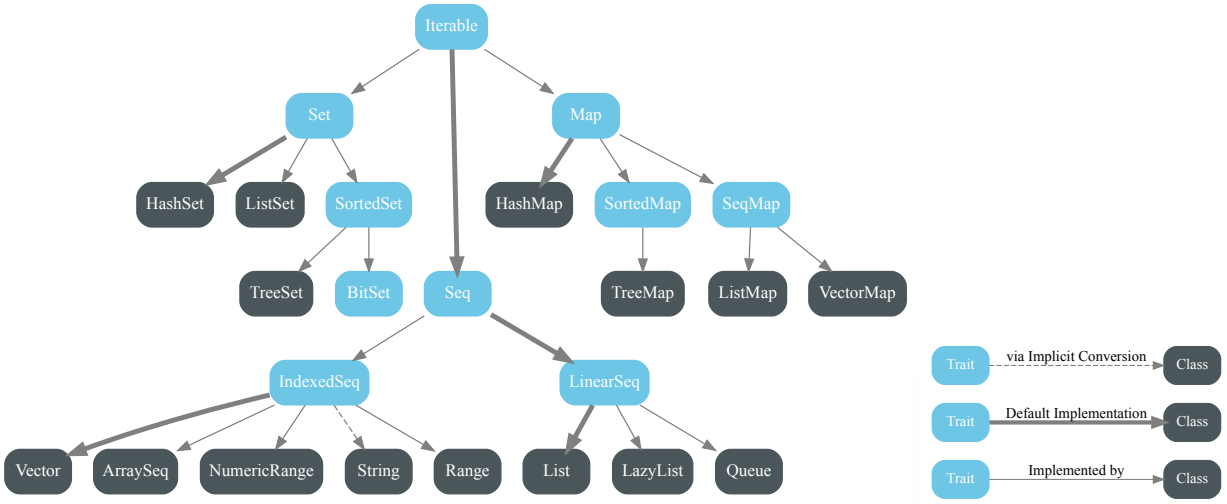


Figure 10.1: Immutable collections tree.

```
val uMap = Map (1 -> 'a', 2 -> 'b', 3 -> 'c', 4 -> 'd')

println (s" Apply method for the List with .apply = ${uList.apply (1)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for the List without .apply = ${uList(1)}")

println (s" Apply method for the Set with .apply = ${uSet.apply(22)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for the Set without .apply = ${uSet(22)}")

println (s" Apply method for Map with .apply = ${uMap.apply(2)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for Map without .apply = ${uMap(2)}")
```

Listing 10.9: Illustration of `apply` method.

10.3 zip and unzip Methods

The `zip` method takes two lists as input and creates a list by pairing elements from the two lists with same index. If the lengths of the two lists are different, then unmatched elements are dropped. The `zip` and `unzip` methods also allow to operate on multiple lists together. Listing 10.10 provides the definitions for `zip`, `unzip` and `zipWithIndex` methods.

```
def zip[B](that: IterableOnce[B]): CC[(A, B)] // CC - arbitrary collection
def unzip[A1, A2](implicit asPair: (A) => (A1, A2)): (CC[A1], CC[A2])
def zipWithIndex: CC[(A, Int)]
```

Listing 10.10: Definitions for zip, unzip and zipWithIndex methods.

The zip and unzip methods are applicable to both immutable as well as mutable collection types. The keyword `CC` in Listing 10.10 represents an arbitrary collection type.

Listing 10.11 illustrates how to apply the zip/unzip methods on two lists with different data types. First, both lists are zipped together after which, the zipped list is unzipped. At the end zipping of list

with its index is also given, where each element in the list is paired with its respective index.

```
val uList1: List[(Char)] = List('a', 'b', 'c', 'd', 'e')
val uList2: List[(Int)] = List(20, 40, 100)

val uList_Zipped = uList1.zip(uList2)
println(s"The zipped list is: $uList_Zipped")

val uList_unZipped = uList_Zipped.unzip
println(s"The unzipped list is: $uList_unZipped")

val uList_indexZip = uList1.zipWithIndex
println(s"The list zipped with its index: $uList_indexZip")

// The output at the terminal
The zipped list is: List((a,20), (b,40), (c,100))
The unzipped list is: (List(a, b, c),List(20, 40, 100))
The list zipped with its index: List((a,0), (b,1), (c,2), (d,3), (e,4))
```

Listing 10.11: Zip and unzip methods.

In Listing 10.11 the zip operation can also be implemented using a different syntax, as described in Listing 10.12. This second syntax is also called *syntactic sugar* or *syntactic convenience*.

```
val uList_Zipped = uList1.zip(uList2)
// is same as
val uList_Zipped = uList1 zip uList2
```

Listing 10.12: Syntactic convenience illustration.

10.4 The reduce Method

The `reduce` is a higher order method available with many Scala collections. This method traverses all the elements in a collection and combines them in *some way* (specified in the argument) to produce the result. The reduce method uses binary operations to combine the elements and can not be applied to an empty collection. It is worth mentioning that the order of applying binary operations does matter. Listing 10.13 provides the definitions for `reduce`, `reduceLeft` and `reduceRight` methods.

```
def reduce[B >: A](op: (B, B) => B): B  // op - operation to be performed
def reduceLeft[B >: A](op: (B, A) => B): B
def reduceRight[B >: A](op: (A, B) => B): B
```

Listing 10.13: Definitions for reduce, reduceLeft and reduceRight methods.

The *syntactic sugar* syntax for reduce method is illustrated in Listing 10.14.

```
uList.reduce((a, b) => a + b)
// is same as
```

```
uList.reduce(_ + _)
// is same as
uList reduce ((a, b) => a + b)
// is same as
uList reduce (_ + _)
```

Listing 10.14: Syntactic convenience illustration for reduce method.

10.4.1 reduce Method: Examples

Let us illustrate the working of reduce method using some examples. Listing 10.15 demonstrates the summation of list elements using `reduce` method. Two instances are given, one with explicit arguments and second using wildcard. Both the cases will produce the same result provided same operation has been applied while invoking the method.

```
// Illustration of reduce method for Lists
val uList = List(1, 2, 3, 4, 5)

val uSum_Explicit = uList.reduce((a, b) => a + b)
println(s"Sum of elements using reduce function explicitly= $uSum_Explicit")

val uSum: Double = uList.reduce(_ + _)
println(s"Sum of elements using reduce function with wildcard = $uSum")
```

Listing 10.15: Illustration of reduce method.

The second example shown in Listing 10.16 instantiates a list and then creates a modified one that pairs each element of the first list with 1. Afterwards reduce method is applied on the compound list and it is explicitly mentioned in the argument that what operation is performed on the first elements of each pair and correspondingly on the second elements. This reduces the paired list to a single pair whose elements are operated to evaluate the average.

```
// source collection
val uList = List(1, 5, 7, 8)

// converting every element to a pair of the form (x,1)
val uList_Modified = uList.map(x => (x, 1))

// adding elements at correspnding positions
val result = uList_Modified.reduce((a, b) => (a._1 + b._1, a._2 + b._2))
val average = (result._1).toFloat / (result._2).toFloat

println("(sum, no_of_elements) = " + result)
println("Average = " + average)
```

Listing 10.16: Reduce method on paired list.

10.4.2 Order of reduce Method

As mentioned before, the order in which the binary operations are applied by the reduce method, matters. There are two methods, that expand on the functionality of the reduce method, namely, `reduceLeft` and `reduceRight`. As their names suggest, the former would yield (a # b) and the latter would be (b # a) where '#' is an arbitrary binary operator. Refer to Listing 10.17 for an application where the order makes a difference in the result. For further details see [3] (Chapters 16 and 25).

```
// Illustration of the order of reduce method
val uList = List(1, 2, 3, 4, 5)

val uSum: Double = uList.reduce(_ - _)
println(s"Difference of elements using reduce function = $uSum")

val uSum_Explicit = uList.reduceLeft(_ - _)
println(s"Difference of elements using reduceLeft = $uSum_Explicit")

val uSum_Explicit1 = uList.reduceRight(_ - _)
println(s"Difference of elements using reduceRight = $uSum_Explicit1")
```

Listing 10.17: The reduceLeft and reduceRight methods.

10.5 fold, foldLeft and foldRight Methods

The `fold` method is used to collapse the elements of a collection using associative binary operator (function). The `fold` method takes two arguments; the initial value and the function. The function itself takes two arguments; the function result from the previous iteration and the current item from the list. For the first iteration, the function result takes the initial value. The Listing 10.18 provides definitions for `fold`, `foldLeft` and `foldRight` methods.

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 // z - initial value,
def foldLeft[B](z: B)(op: (B, A) => B): B  // op - operation to be performed
def foldRight[B](z: B)(op: (A, B) => B): B
```

Listing 10.18: Definition of fold, foldLeft and foldRight methods.

We repeat the example in Listing 10.16, by only replacing the `reduce` method with `fold` method. The result is provided in Listing 10.19

```
// source collection
val uList = List(1, 5, 7, 8)

// converting every element to a pair of the form (x,1)
val uList_Modified = uList.map(x => (x, 1))

// adding elements at correspnding positions
val result = uList_Modified.fold(0,0)((a, b) => (a._1 + b._1, a._2 + b._2))
val average = (result._1).toFloat / (result._2).toFloat
```

```
println("(sum, no_of_elements) = " + result)
println("Average = " + average)

// The result at the terminal
(sum, no_of_elements) = (21,4)
Average = 5.25
```

Listing 10.19: The fold method to evaluate list average.

Other variants of fold method are `foldLeft` and `foldRight`. The primary difference among these methods is the order in which the operation is performed over the collection. The `foldLeft` starts on the left as the first element and iterates towards right, while the `foldRight` starts on the right. The `fold` method does not follow a particular order but has some constraints on the start value. For instance, the start value should be a supertype of the object being folded. A second constraint is that the initial value must be neutral, that is it should not modify the result. Listing 10.20 compares the operation of three methods.

```
// Illustration of the order of fold method
val uList = List(1, 2, 3, 4, 5)

val uSum_fold = uList.fold(0)(_ - _)
println(s"Elements difference using fold method = $uSum_fold")

val uSum_foldLeft = uList.foldLeft(0)(_ - _)
println(s"Elements difference using foldLeft method = $uSum_foldLeft")

val uSum_foldRight = uList.foldRight(0)(_ - _)
println(s"Elements difference using foldRight method = $uSum_foldRight")

// The output at the terminal
Elements difference using fold method = -15
Elements difference using foldLeft method = -15
Elements difference using foldRight method = 3
```

Listing 10.20: The fold method and its variants.

10.5.1 Processing Multiple Lists

In Scala, multiple lists can be operated simultaneously. Listing 10.21 shows such an example where two lists with different lengths are instantiated and then mapped to a tuple using the zipped method. These tuples can then be operated with methods like `map` and `max`. Note that the max method uses only the elements from the first list to find the max value and then returns the respective pair.

```
val uList1: List[(Int)] = List(3, 2)
val uList2: List[(Int)] = List(20, 40, 100)

// Processing multiple lists
val resultProduct = (uList1, uList2).zipped.map(_ * _)
```

```
val resultCount = (uList1, uList2).zipped.max

println(s"The product of two lists: $resultProduct")
println(s"The max element: $resultCount")
```

Listing 10.21: Processing multiple lists.

10.6 Illustrations from Rocket Chip

Different methods discussed above are further illustrated using selected examples from Rocket[1] chip generator. The first example defines an exception (`checkExceptions`) and hazard (`checkHazards`) checking methods as outlined in Listing 10.22. The `map` and `reduce` methods are applied, to a Seq of Tuple2, to find out if there is an exception/hazard or not? Also note the return types of these user defined methods.

```
// Exception checking in rocket core
def checkExceptions(x: Seq[(Bool, UInt)]): (Bool, UInt) =
// Seq of exception flags, cause of exception
(x.map(_._1).reduce(_||_), PriorityMux(x))

// Hazard checking in rocket core
def checkHazards(targets: Seq[(Bool, UInt)], cond: UInt => Bool): Bool =
targets.map(h => h._1 && cond(h._2)).reduce(_||_)
```

Listing 10.22: Exception and hazard checking.

The second example is the register file implementation with multiple read ports and one write port. The implementation also resolves the read after write hazard by forwarding as illustrated in Listing 10.23.

```
class RegFile(n: Int, w: Int, zero: Boolean = false) {
   val rf = Mem(n, UInt(width = w))
   private def access(addr: UInt) = rf(~addr(log2Up(n)-1,0))
   private val reads = ArrayBuffer[(UInt,UInt)]()
   private var canRead = true

   def read(addr: UInt) = {
      require(canRead)
      reads += addr -> Wire(UInt())
      reads.last._2 := Mux(Bool(zero) && addr === UInt(0), UInt(0), access(
       addr))
      reads.last._2
   }
   def write(addr: UInt, data: UInt) = {
      canRead = false
      when (addr =/= UInt(0)) {
         access(addr) := data
```

[1] https://github.com/chipsalliance/rocket-chip

```
            for ((raddr, rdata) <- reads)
            when (addr === raddr) {
                rdata := data
            } // check for forwarding
        }
    }
}
```

Listing 10.23: Exception and hazard checking.

10.7 Exercises

Exercise 1: Write Scala program that implements the function $y = ax^2 + bx + c$, where $a = 3,\ b = 5$ and $c = 7$ (this is similar to what has been done in Listing 10.4). Create a list of integers in the range $-3 \leq x \leq 3$ and use the defined function to map its elements to another list. Print the mapped list and verify the results. (Try to use wildcard wherever possible.)

Exercise 2: Zip the two lists created in Exercise 1 and then zip the resulting list with its index. A list with three elements per pair is created in the following format: $(x, f(x), index)$. Find the mean value of $f(x)$ and store the respective pair to a variable *mean*. Refer to Listings 10.11, 10.16 and 10.21 for possible hints.

Exercise 3: Write a Scala program that takes in a vector as an integer list and calculates its Euclidean norm. Recall that this is also termed as the magnitude of a vector and is evaluated as given below.

$$||\vec{u}||_2 = \left(\sum_{i=1}^{N} |u_i|^2\right)^{\frac{1}{2}} = \sqrt{u_1^2 + u_2^2 + ... + u_N^2}$$

Experiment 11

Scala II

Objective

Primarily focus of this lab is to introduce different advanced concepts in Scala. We start with `map` and `flatMap` higher order methods, which help in writing generators. Then different types of classes and objects in Scala are discusses, elaborating the importance as well as their use.

11.1 Map, map and flatMap

The `Map` is, by default, an immutable collection in Scala. As discussed previously, it is a collection of key-value pairs, which is similar to a dictionary. Key-value pairs, in Map, can have an arbitrary data type. However, once the data type has been used for a key and the associated value, then its consistency must be maintained throughout.

On the other hand `map` and `flatMap` are methods that can be applied to different collections including Map collection. The `map` is a functional mapping applied to each element of the collection. Similarly, the `flatMap` is also a functional mapping applied to each element but it flattens the results as will be illustrated later with examples. Listing 11.1 provides the definitions for `map` and `flatMap` methods.

```
def map[B](f: (A) => B): CC[B] // CC - arbitrary collection
def flatMap[B](f: (A) => IterableOnce[B]): CC[B]
```

Listing 11.1: Definitions for map and flatMap methods.

Listing 11.2 illustrates the use of `map` method applied to a list. The function used for mapping can be mentioned either explicitly or can have the form of a user defined function.

```
// An example list
val uList = List(1, 2, 3, 4, 5)

// map method applied to List
val uList_Twice = uList.map( x => x*2 )
println(s"List elements doubled = $uList_Twice")

// Applying map to List using user defined method
def f(x: Int) = if (x > 2) x*x else None
val uList_Squared = uList.map(x => f(x))
println(s"List elements squared selectively = $uList_Squared")
```

```
// The output at the terminal is given below
List elements doubled = List(2, 4, 6, 8, 10)
List elements squared selectively = List(None, None, 9, 16, 25)
```

Listing 11.2: Illustration of map method.

Next we compare and contrast the working of map and flatMap methods. For that purpose, we apply the map and flatMap to the same list by employing the user defined function g(v:Int) as discussed in Listing 11.3. The flatMap not only flattens the list but also eliminates any nonexistent (empty) entries from the resulting list as can be seen from the illustration in Listing 11.4.

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

def g(v:Int) = List(v-1, v, v+1)
val uList_Extended = uList.map(x => g(x))
println(s"Extended list using map = $uList_Extended")

val uList_Extended_flatmap = uList.flatMap(x => g(x))
println(s"Extended list using flatMap = $uList_Extended_flatmap")

// The output at the terminal is
Extended list using map = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4),
    List(3, 4, 5), List(4, 5, 6))

Extended list using flatMap = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5,
     6)
```

Listing 11.3: Illustration of map and flatMap methods.

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

// Applying map and flatMap to List with builtin Options class
def f(x: Int) = if (x > 2) Some(x) else None
val uList_selective = uList.map(x => f(x))
println(s"Selective elements of List with .map = $uList_selective")

val uList_selective_flatMap = uList.flatMap(x => f(x))
println(s"Selective elements of List with .flatMap =
    $uList_selective_flatMap")

// Output at the terminal
Selective elements of List using .map = List(None, None, Some(3), Some(4),
    Some(5))
Selective elements of List using .flatMap = List(3, 4, 5)
```

Listing 11.4: Illustration of map and flatMap method.

Finally, we illustrate the use of map and flatMap with the Map type collections. Listing 11.5 illustrates the use of map and flatMap with a user defined Map 'uMap'.

```
// An example Map using (key, value) pairs
val uMap = Map('a' -> 2, 'b' -> 4, 'c' -> 6)

// Applying .mapValues to Map
val uMap_mapValues = uMap.mapValues(v => v*2)
println(s"Map values doubled using .mapValues = $uMap_mapValues")

def h(k:Int, v:Int) =  Some(k->v*2)

// Applying .map to Map
val uMap_map = uMap.map {
    case (k,v) => h(k,v)
}
println(s"Map values doubled using .map = $uMap_map")

// Applying .flatMap to Map
val uMap_flatMap = uMap.flatMap {
    case (k,v) => h(k,v)
}
println(s"Map values doubled using .flatMap = $uMap_flatMap")

// The output at the terminal
Map values doubled using .mapValues = Map(a -> 4, b -> 8, c -> 12)
Map values doubled using .map = List(Some((97,4)), Some((98,8)), Some
    ((99,12)))
Map values doubled using .flatMap = Map(97 -> 4, 98 -> 8, 99 -> 12)
```

Listing 11.5: Illustration of map and flatMap methods applied to Map collection.

Listing 11.6 illustrates different syntax variants using syntactic sugar, for map method, while generating the same result.

```
val uList = List(1, 2, 3, 4, 5)

val uList_mapped1 = uList map (x => x * 2) map (x => x + 3)
val uList_mapped2 = uList.map(x => x * 2).map(x => x + 3)
val uList_mapped3 = uList.map(_ * 2).map(_ + 3)
val uList_mapped4 = uList.map(x => x * 2).map(_ + 3)
val uList_mapped5 = uList.map(_ * 2) map(x => x + 3)
```

Listing 11.6: Syntactic sugar illustration for map method.

11.2 More on Classes and Objects

Below are some key attributes applicable to classes and objects in Scala.

- Classes are blueprints for creating objects which are real. They can contain methods, values, variables, types, objects, traits etc. Once a class is defined, then you can create objects from the class with the keyword `new`.
- A Public modifier is not allowed, but, `Private`, `Abstract` and `Protected` modifiers are permitted.
- Source file can contain many classes and there is no need to match file name with class name.

11.2.1 Abstract Classes

The `abstract` modifier with Class signifies that the class may have *abstract members*. Abstract classes can not be instantiated. In an abstract class, the abstract members:

- do not have an implementation
- can have constructor parameters, but a trait cannot
- do not require `abstract` modifier with the method name, when defined inside an abstract class

In Listing 11.7 an abstract class is defined. The method 'contents' is an abstract member of this class, though the keyword `abstract` is not used as a modifier in the method definition. The `Data_buffer_fill` is a sub or derived class that implements the method contents and as a result it is a *concrete class*.

```
abstract class Data_buffer {
    def contents: Array[String]
}
```

Listing 11.7: Abstract class declaration.

```
class Data_buffer_fill(data: Array[String]) extends Data_buffer {
    def contents: Array[String] = data
}
```

Listing 11.8: Concrete class inherited from an abstract class.

11.2.2 Companion Objects

An object defined using `object` keyword and having only one instance is called *singleton object*. When a singleton object has the same name as that of a class, it is called the companion object of that class while the corresponding class is a *companion class*. A singleton object that does not have a companion class is a *standalone object*.

Scala does not permit static methods to be declared in classes. Scala defines companion objects with the same name as that of the class. Static methods are placed in the companion object. A call to `Class.method` is actually a call to the method in the companion object. Listing 11.9 and 11.10 provide an illustration of the companion Control object and the corresponding companion Control class. It is important to note that the companion object and its companion class should be placed in the same file.

```
object Control {


    val default =
    //                                                          kill                          wb_en  illegal?  en_rs2
    //            pc_sel  A_sel   B_sel  imm_sel   alu_op   br_type |  st_type ld_type wb_sel  | csr_cmd | en_rs1 |
    //              |       |       |      |         |          |    |     |       |       |    | |       | |    |
    List(PC_4,   A_XXX,  B_XXX, IMM_X, ALU_XXX   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, Y, N,   N)
    val map = Array(
    LUI   -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_COPY_B, BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,   N),
    AUIPC -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_ADD   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,   N),
    JAL   -> List(PC_ALU, A_PC,   B_IMM, IMM_J, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, N,   N),
    JALR  -> List(PC_ALU, A_RS1,  B_IMM, IMM_I, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, Y,   N),
```

Listing 11.9: Companion Control object of Control class.

```
class Control extends Module {
    val io = IO(new ControlSignals)
    val ctrlSignals = ListLookup(io.inst, Control.default, Control.map)

    // Control signals for Fetch
    io.pc_sel    := ctrlSignals(0)
    io.inst_kill := ctrlSignals(6).toBool

    // Control signals for Execute
    io.A_sel   := ctrlSignals(1)
    io.B_sel   := ctrlSignals(2)
    ...
```

Listing 11.10: Control class.

The apply method, implemented in companion objects, is used to construct objects without using the keyword new. An apply method takes construction parameters and constructs an object. Correspondingly, an unapply method takes an object and extracts values (parameters) from it. An extractor is an object with an unapply method. Listing 11.11 illustrates the use of unapply method for parameter extraction from an object.

```
class uModule(val name: String, val bitWidth: Int)

// companion object
object uModule {
  def apply(name: String, bitWidth: Int): uModule = new uModule(name,
      bitWidth)

  def unapply(mod: uModule): Option[(String, Int)] = {
    if (mod.bitWidth == 0) None
    else Some((mod.name, mod.bitWidth))
  }
}

// Using the apply method
val objA = uModule("ALU", 32)
```

```
// Extractor using unapply method
val uModule(module_name, module_bitW) = objA
println("Module name is: " + module_name)

// output at the terminal is
Module name is: ALU
```

Listing 11.11: Illustration of unapply method. format

11.2.3 Case Class

A `case class` is like a regular class, but has some distinct features. It adds a factory method with the name of the class. This factory method manages the construction of object and does not require the use of keyword `new` when instantiating. All arguments in the parameter list of a case class implicitly get a `val` prefix.

- The class parameters in case classes are promoted to fields and are sensible to `toString`
- Methods `equals` and `hashcode` are included with case classes making them important for collections
- case classes come with companion objects by default

The compiler adds a *copy* method to the class to make modified copies of the class. Similar to case classes we have `case objects`.

Listing 11.12 provides and example showing the implementation of case class. It is also illustrated that the arguments are of `val` type, which does not permit reassignment. The use of copy method to make multiple modified copies with different parametric values is illustrated in Listing 11.13.

```
case class register(addr:Int, init: Int)

var c = register(100, 11001100)

// Display the register
println("Reg Addr: " + c.addr + " Initial value: " + c.init)

// the output on the terminal
Reg Addr: 100    Initial value: 11001100

// when tried to update the address
c.addr = 15

// the following error is encountered
reassignment to val error
```

Listing 11.12: An example illustration of Reassignment error.

```
case class register(addr:Int, init: Int)
```

```
var c = register(100, 11001100)
var d = c.copy(addr = 104)

// Display the register
println("Reg Addr: " + c.addr + " Initial value: " + c.init)
println("Reg Addr: " + d.addr + " Initial value: " + d.init)

// the output on the terminal
Reg Addr: 100    Initial value: 11001100
Reg Addr: 104    Initial value: 11001100
```

Listing 11.13: Illustration of copy method.

One of the key advantages of case classes is that they support pattern matching and also create an unapply method that provides access to all of the class parameters. This is illustrated in Listing 11.14.

```
case class uModule(name: String, bitWidth: Int)

// pattern matching using unapply method of case class
def matchObject(obj: uModule) = {
   val result = obj match {
      case uModule(name, 32) => println("Module name is: " + name)
      case uModule(name, 16) => println("Module name is: " + name)
      case _                 => println(None)
   }
}
// instantiate different modules
val objA = uModule("ALU", 32)
val objI = uModule("Imm", 32)
val objB = uModule("Branch", 16)
val objM = uModule("Mul", 64)

matchObject(objI)

// output at the terminal
Module name is: Imm
```

Listing 11.14: Pattern matching using case class.

11.3 Illustrations from Rocket Chip

The first example determines the bypass source using map method and is given in Listing 11.15. An indexed sequence is used to form a collection of bypass sources.

```
// Bypass source ID
val bypass_sources = IndexedSeq(
(Bool(true), UInt(0), UInt(0)), // treat reading x0 as a bypass
(ex_reg_valid && ex_ctrl.wxd, ex_waddr, mem_reg_wdata),
```

```
(mem_reg_valid && mem_ctrl.wxd && !mem_ctrl.mem, mem_waddr, wb_reg_wdata),
(mem_reg_valid && mem_ctrl.wxd, mem_waddr, dcache_bypass_data))

val id_bypass_src = id_raddr.map(raddr => bypass_sources.map(s => s._1 && s.
    _2 === raddr))
```

Listing 11.15: Determining the source of bypass.

The generation of control signals, using decode method, is illustrated in Listing 11.16. The use syntactic sugar syntax for `zip` and `map` methods can be noticed, when assigning the generated control signals to the IO signals.

```
def decode(inst: UInt, table: Iterable[(BitPat, List[BitPat])]) = {

    val decoder = DecodeLogic(inst, default, table)
    val sigs = Seq(legal, fp, rocc, branch, jal, jalr, rxs2, rxs1, scie,
    sel_alu2, sel_alu1, sel_imm, alu_dw, alu_fn, mem, mem_cmd, rfs1,
    rfs2, rfs3, wfd, mul, div, wxd, csr, fence_i, fence, amo, dp)

    sigs zip decoder map {
        case(s, d) => s := d
    }
    this
}
```

Listing 11.16: Wiring of control sugnals.

11.4 Exercises

Exercise 1: Design an FSM by using a companion object. Put static functionality in companion object i.e. `enum` will be placed inside the `object` instead of `class` then use this to develop your FSM code inside the class.

Exercise 2: Create a shallow and a deep copy of an object of a case class given in the Listing 11.13. Find out whether the copy method available with case class gives a shallow or a deep copy.[1]

Exercise 3: Refer to the Listing 11.2,11.3,11.4,11.5. Find out which other collections support `map` and `flatMap` methods and go through their illustrations.

Problem 4: Refer to Exp 9, Problem 5. Solve the same problem but using `map` and `flatMap`.

[1]For further details see: https://stackoverflow.com/questions/52966711/scala-case-class-uses-shallow-copy-or-deep-copy

Experiment 12

Design Project

Objective

The main goal of this project is to have a hands-on experience in designing a processor using Chisel. The project covers fundamental areas in the processor design based on RISC V architecture. The main objective of this exercise is to learn the transformation of a micro-architecture to its corresponding Chisel based implementation that can be tested against the anticipated performance measures.

12.1 Project Description

The key objective of this project is to enable you to be able to transform the given RISC V Instruction Set Architecture (ISA) specifications to a working hardware prototype that can not only be tested for performance analysis but is also reusable due to modular design. The suggested project implementation will comply the following.

- RV32I base instruction set with machine mode only
- Harvard bus architecture with separate data and instruction buses
- Detection of data hazards and their resolution
- User-level ISA Spec. 2.2 to be followed
- Privileged architecture Spec. 1.11 to be followed

12.2 Processor Core Modules

Its a good practice to adopt modular design when implementing a microprocessor architecture. The micro-architecture naturally provides that opportunity for modular design. The processor will have the following key modules.

- ALU
- Conditional Branch
- Immediate value generation module
- Register file
- Control and Status Registers (CSRs)
- Instruction and data memory interfaces
- The controller

- The datapath

The ALU, conditional branch, immediate value generation, register file and CSR file are lumped modules based on their functionality. Instruction and data memory interfaces are required for memory connectivity. The two memory modules are not part of the processor implementation, however, they should be implemented for functional testing of the processor. Finally, the controller and the datapath can be considered as distributed modules from the functionality implementation viewpoint.

Some of the above mentioned modules have already been developed and tested as part of previous laboratory sessions. Use those modules to start with. It is recommended to test each module independently, before its integration.

12.3 Microarchitecture

Microarchitecture provides the necessary details required for the implementation of the processor (refer to Figures 12.4 and 12.5 for details). We will be implementing a three stage pipelined architecture with the following stages.

- Instruction Fetch
- Decode & Execute
- Memory & Writeback

In this implementation, there will be no branch prediction. The code and data memories will be based on simple synchronous/asynchronous memories discussed previously.

12.4 Fetch Stage

The fetch stage will be responsible of fetching the instructions from the code memory and depending on the control signals, the fetched instruction will be either stalled or will be fed to the decode stage. It is recommended to use synchronous memory for implementing code memory. However, if you can show any performance improvements, you are free to use asynchronous memory. The fetch stage will not implement any branch prediction. The block diagram shown in Figure 12.1 suggests a possible implementation of fetch stage.

12.5 Decode and Execute Stage

This stage will be responsible for decoding of the instructions to determine the type of operation to be performed and the corresponding operands required. The operands are either registers or immediate values. The control signals (for all stages) will also be generated at this point of time. Furthermore before the execution can happen, data hazards detection and forwarding (if required and possible) should be performed as part of implementation of this stage. Figure 12.2 illustrates the microarchitecture for the decode and execute stage.

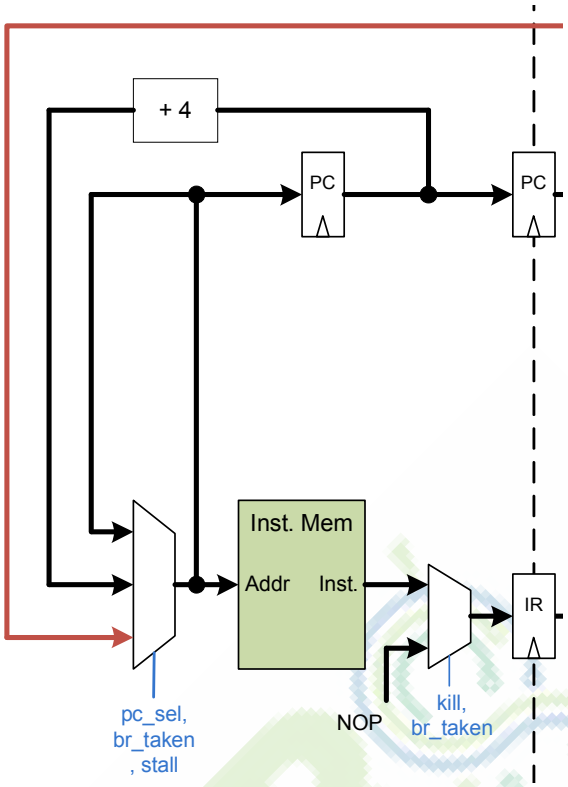


Figure 12.1: The fetch stage.

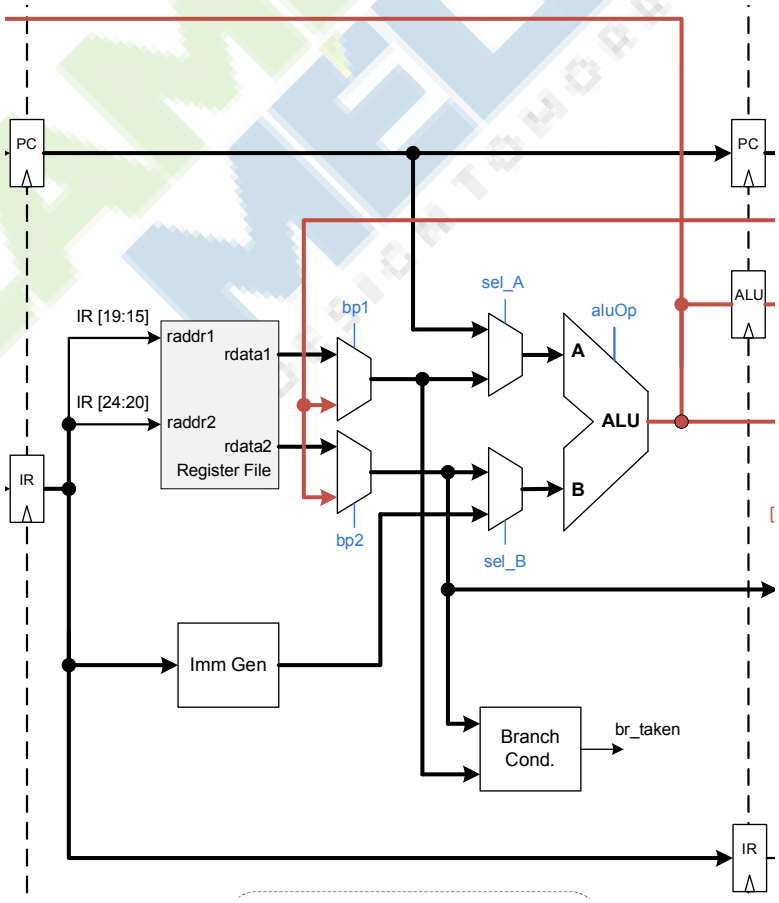


Figure 12.2: Decode and execute stage.

12.6 Memory and Writeback Stage

We will use synchronous memory for data. The CSR register-file read and write operations will also be performed during this stage. In addition, the CSR module will also implement the necessary hardware that will be responsible for handling of the external interrupts. Block diagram in Figure 12.3 shows the micro architecture for memory and writeback stage.

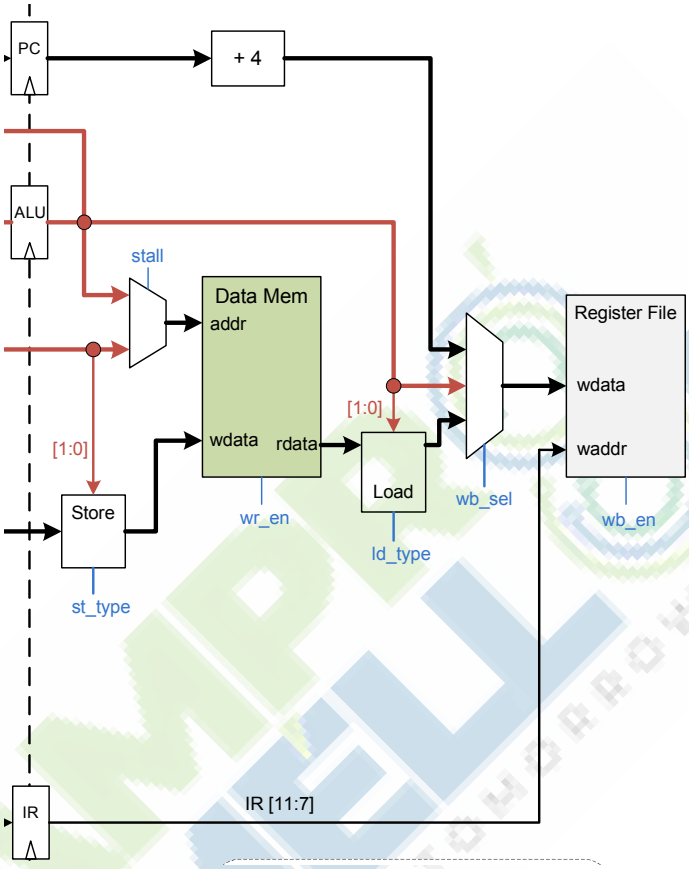


Figure 12.3: Memory and writeback stage.

12.6.1 RISC V Privilege Levels

Privilege levels are used to provide protection among different components of the software stack. At a given time instant, a RISC-V hardware thread (hart) is running at some privilege level. Privilege level is encoded as a mode in one or more CSRs. Different privilege levels and their combinations supported by the Privileged Architecture Specifications [5] are tabulated in Table 12.1 and Table 12.2 respectively. As mentioned above, the current implementation only aims for the highest privilege level i.e. machine level (M) with privilege level 3.

Table 12.1: RISC V privilege levels.

Level	Encoding	Name	Abbreviation
0	00	User/application level	U
1	01	Supervisor level	S
2	10	Reserved	
3	11	Machine level	M

Table 12.2: Combinations of privilege levels that can be supported.

No. of Level	Supported Modes	Usage
1	M	For simple embedded systems
2	M, U	For secure embedded systems
3	M, S, U	For systems running Linux type operating systems

12.6.2 Machine Mode Registers

The RISC-V ISA has allocated 12-bit address space for CSRs. Of these 12 bits, the top 4 bits of CSR address space (csr[11:8]) are allocated to encode read/write accessibility of the CSRs based on current privilege level. Of these four bits, top two bits (csr[11:10]) indicate whether the register accessibility is read/write (00, 01, or 10) or read-only (11), while the next two bits (csr[9:8]) encode the lowest privilege level allowed to access the CSR [5].

The machine mode registers are divided into multiple subgroups based on their usage and associated addressing. Table 12.3 lists the grouping of machine mode CSRs along with their addressing.

Table 12.3: RISC-V machine-level CSR groups and addressing.

(csr[11:10])	(csr[11:10])	Hexadecimal Address	Description
00	11	0x30x	Machine trap/interrupt setup
00	11	0x32x, 0x33x	Machine counter setup
00	11	0x34x	Machine trap handling
00	11	0x3Ax, 0x3Bx	Machine memory protection
01	11	0x7Ax	Debug/Trace registers (shared with Debug mode)
01	11	0x7Bx	Debug mode registers
10	11	0xBxx	Machine counter/timers
11	11	0xF1x	Machine information registers

Of these machine mode CSRs, we will only concentrate on machine trap/interrupt setup and trap handling registers. These registers should be implemented to provide support for interrupts. The CSRs from machine trap/interrupt setup subgroup are listed in Table 12.4, while machine trap handling CSRs are listed in Table 12.5. We have also provided details for machine information CSRs in Table 12.6 for reference purpose only. The bitfield details for these registers can be found in [5].

Table 12.4: Machine trap/interrupt setup registers.

Access Privilege	Address	Register Name	Description
RW	0x300	mstatus	Machine status register.
RW	0x301	misa	ISA and extensions.
RW	0x302	medeleg	Machine exception delegation register.
RW	0x303	mideleg	Machine interrupt delegation register.
RW	0x304	mie	Machine interrupt-enable register.
RW	0x305	mtvec	Machine trap-handler base address.
RW	0x306	mcounteren	Machine counter enable.

To support interrupts with minimum functionality, you are required to implement the following registers.

- Machine trap/interrupt setup registers: `mstatus`, `misa`,`mie`,`mtvec`
- Machine trap handling registers: `mscratch`, `mepc`,`mcause`,`mtval`, `mip`

Table 12.5: Machine trap handling registers.

Access Privilege	Address	Register Name	Description
RW	0x340	mscratch	Scratch register for machine trap handlers.
RW	0x341	mepc	Machine exception program counter.
RW	0x342	mcause	Machine trap cause.
RW	0x343	mtval	Machine bad address or instruction.
RW	0x344	mip	Machine interrupt pending.

Table 12.6: Machine information registers.

Access Privilege	Address	Register Name	Description
RO	0xF11	mvendorid	Vendor ID.
RO	0xF12	marchid	Architecture ID.
RO	0xF13	mimpid	Implementation ID.
RO	0xF14	mhartid	Hardware thread ID.

12.7 Preparing the Executable for Testing

Support for interrupts is required as part of this RISC V based processor implementation project. To test this functionality, we need to write user programs that cater the need for interrupt support. Specifically, interrupt vector table and interrupt service routine (ISR) should be implemented as part of user program. This software based functionality is usually implemented in separate assembly language files called 'startup.s' and 'isr.s' (also called system files). A sample 'startup.s' is provided in Listing 12.1 and 'isr.s' in Listing 12.2.

```
.equ CSR_MSTATUS, 0x300
.equ MSTATUS_MIE, 0x00000008
.equ CSR_MTVEC,   0x305

 # Main interrupt vector table entries
.global vtable
.type vtable, %object
.section .text.vector_table,"a",%progbits

# this entry is to align reset_handler at address 0x04
  .word    0x00000013
  j        reset_handler
  .align   2
vtable:
  j        default_interrupt_handler
  .word    0
  .word    0
  j        msip_handler
  .word    0
  .word    0
  .word    0
  j        mtip_handler
  .word    0
  .word    0
  .word    0
  .word    0
  .word    0
  .word    0
  .word    0
```

```
  .word     0
  j         user_handler
  .word     0
  .word     0

# Weak aliases to point each exception handler to the
# 'default_interrupt_handler', unless the application defines
# a function with the same name to override the reference.

  .weak msip_handler
  .set  msip_handler, default_interrupt_handler
  .weak mtip_handler
  .set  mtip_handler, default_interrupt_handler
  .weak user_handler
  .set  user_handler, default_interrupt_handler

# Assembly 'reset handler' function to initialize core CPU registers.
.section .text.default_interrupt_handler,"ax",%progbits

.global reset_handler
.type reset_handler,@function

reset_handler:
# Set mstatus bit MIE = 1 (enable M mode interrupts)
  li      t0, 8
  csrrs   zero, CSR_MSTATUS, t0

# Load the initial stack pointer value.
  la   sp, _sp

# Set the vector table's base address.
  la   a0, vtable
  addi a0, a0, 1
  csrw CSR_MTVEC, a0

# Call user 'main(0,0)' (.data/.bss sections initialized there)
  li   a0, 0
  li   a1, 0
  call main

# A 'default' handler, in case an interrupt triggers without its handler defined
default_interrupt_handler:
  j default_interrupt_handler
```

Listing 12.1: Building an executable for instruction memroy.

```
# RISC-V Interrupt Service Routines (ISRs)
# ALL supported ISRs should be put here

.section .text.isr

# User interrupt handler
.globl user_handler
user_handler:
  nop
  # you can call user ISR here and then return using 'mret'
  mret
```

Listing 12.2: Building an executable for instruction memroy.

Using the system files we can write a user program and build an executable that can be loaded to instruction memory for testing the processor. Listing 12.3 provides a simple user program written for 'main.c' to perform functional testing of the processor.

```
int add(int x, int y) {
  return x+y;
}

int main(void) {
    // declare some variables
    int  x = 123, y = 987, z = 0;

    // call the user function
    z = add(x,y);

    // endless loop
    while(1){}
}
```

Listing 12.3: Example user program in 'main.c'.

12.7.1 Building the Example Program

The Listing 12.4 below shows the compilation and building process for an executable that can be loaded to the instruction memory. Specifically, three source files, 'main.c', 'startup.s' and 'isr.s' are compiled and linked to build the 'program.elf' output, which is used to construct 'main.bin'. This binary file is converted to 'main.hex' as well as 'main.txt' files. Later we load the output file 'main.txt' to the instruction memory (for simulation based processor testing). The steps to load 'main.txt' to instruction memory are discussed in Exp 8 (see Section 8.1.5).

```
riscv64-unknown-elf-gcc -c -o build/main.o src/main.c -march=rv32i -mabi=ilp32

riscv64-unknown-elf-as -c -o build/startup.o src/startup.s -march=rv32i -mabi=ilp32

riscv64-unknown-elf-as -c -o build/isr.o src/isr.s -march=rv32i -mabi=ilp32

riscv64-unknown-elf-gcc -o build/program.elf build/startup.o build/isr.o build/main.o
    -T linker.ld -nostdlib -march=rv32i -mabi=ilp32

riscv64-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text*
   build/program.elf build/main.bin

hexdump build/main.bin > build/main.hex

python maketxt.py build/main.bin > build/main.txt

riscv64-unknown-elf-objdump -S -s build/program.elf > build/program.dump
```

Listing 12.4: Building an executable for instruction memroy.

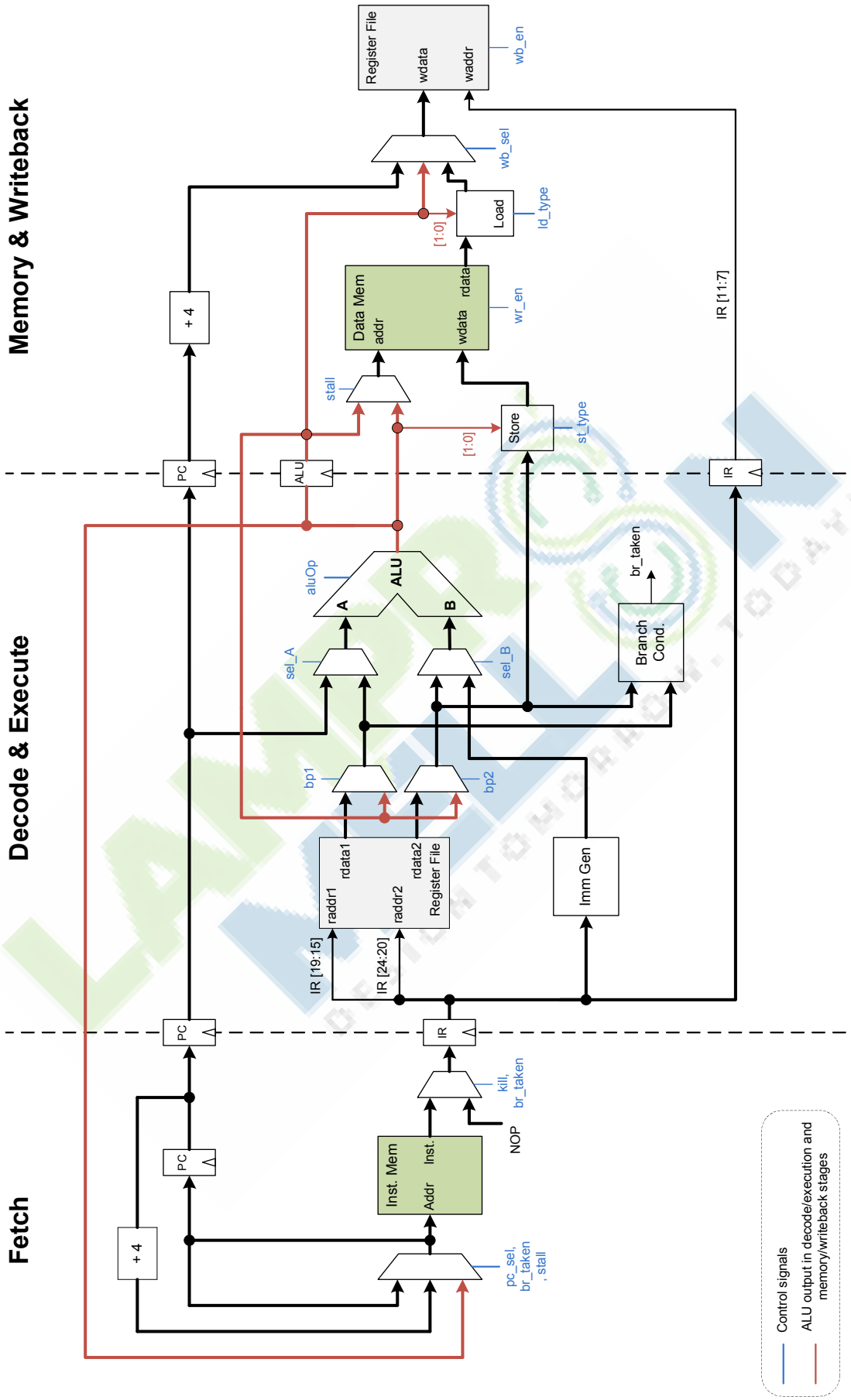


Figure 12.4: Micro-architecture diagram without CSRs.

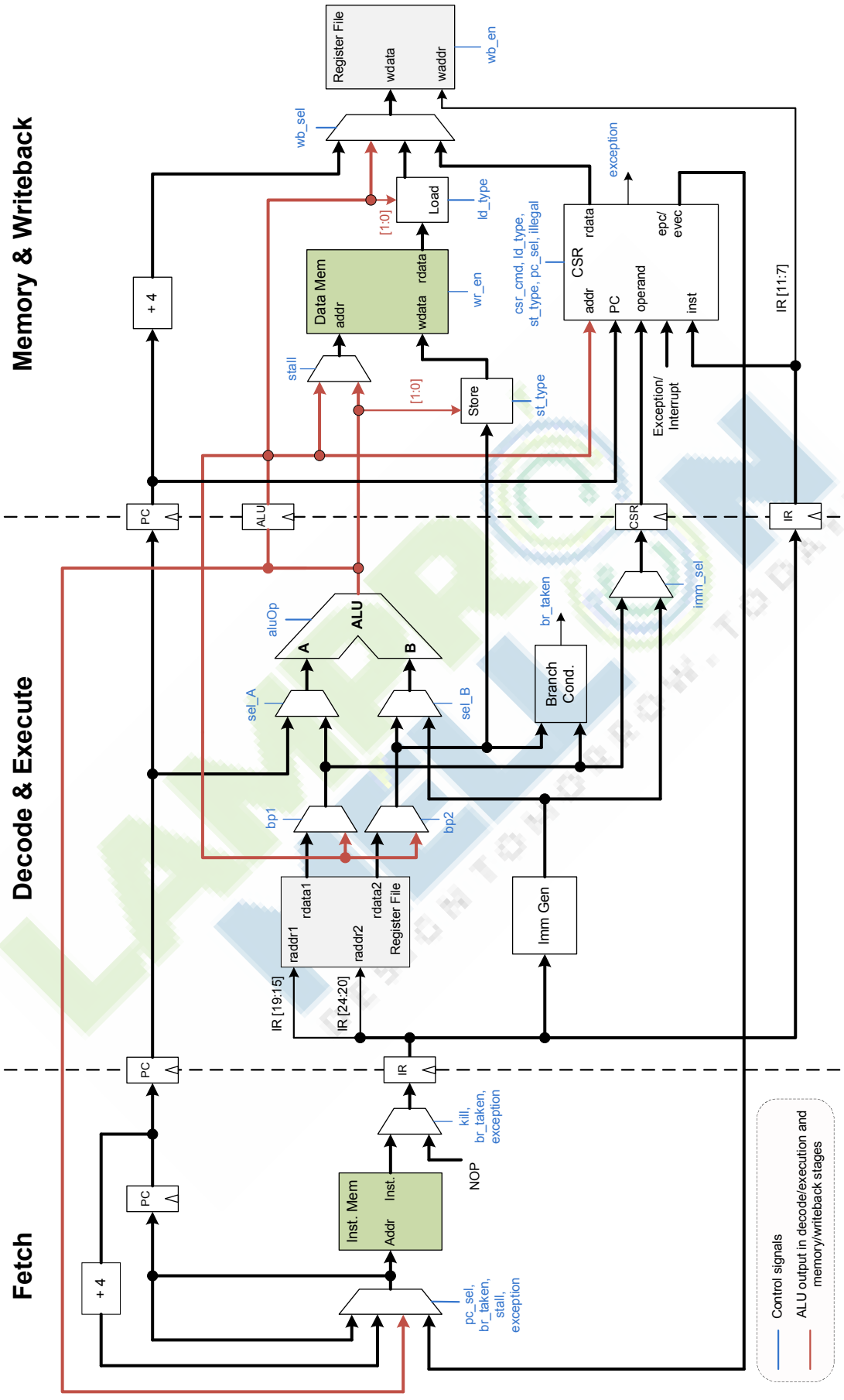


Figure 12.5: Micro architecture with CSRs.

Experiment 13

Scala III

Objective

First understand the working and then learn to use Scala traits, inheritance and linear flattening. In addition, explore advanced parameterization and Scala partial functions.

13.1 Scala Traits and Inheritance

Traits are a fundamental code reuse block in Scala. A trait encapsulates method and field definitions. One key difference is that traits do not have a constructor. A class can mix-in any number of traits. Traits are commonly used for widening thin interfaces to rich interfaces and to define stackable modifications. Traits allow to inherit from multiple class-like constructs. One important difference between traits and multiple inheritance is the interpretation of `super`.

With multiple inheritance, the method called by a super call can be determined right where the call appears. However, with traits, the method called is determined by a procedure called linearization of classes and traits that are mixed into a class. When you call a method on a class with mixins, the method in the trait furthest to the right is called first

13.1.1 When to use traits

When implementing a collection of behavior and the behavior is not going to be reused, then it is preferred to make it a concrete class. If the behavior will be inherited by other classes, then make it an abstract class. However, when the behavior may be used by multiple different (possibly unrelated) classes, then it is better to make it a trait.

13.2 Linear flattening

Scala avoids multiple inheritance by using a technique called linearization. Linearization flattens calls to super classes and effectively solves the diamond problem resulting from multiple inheritance. An example subclass named 'DerivedClass' inheriting from multiple traits and 'BaseClass' is given in Listing 13.1. From the listing it can be observed that no super class was added to the BaseClass. As per Scala hierarchy, for any Scala class the default super class is scala.AnyRef and similarly scala.AnyRef extends scala.Any.

In the example program, the BaseClass has a function 'print', which is overridden by each of the traits and the DerivedClass. Each trait as well as the derived class also make a call to the print function of the super class.

```
class BaseClass{
    def print{
        println("Base Class")
    }
}

trait Trait1 extends BaseClass{
    override def print(){
        println("Trait 1")
        super.print
    }
}

trait Trait2 extends BaseClass{
    override def print(){
        println("Trait 2")
        super.print
    }
}

trait Trait3 extends Trait1{
    override def print(){
        println("Trait 3")
        super.print
    }
}

// Derived class extending base class and mixins with traits
class DerivedClass extends BaseClass with Trait2 with Trait3{
    override def print(){
        println("Derived Class")
        super.print
    }
}

val printFun = new DerivedClass
printFun.print
```

Listing 13.1: Linear falttening example program.

Since the Derived Class is inherited from multiple traits and a BaseClass as a result the compiler will linearize it. Next we explain how its done. The block diagram in Figure 13.1 shows the inheritance structure for the example program in Listing 13.1.

Step 1: To linearize this inheritance structure we traverse all possible paths from the 'DerivedClass' to the top class. Note that we have three class/traits declared as super types for the 'DerivedClass', which are 'BaseClass', Trait2 and Trait3. This can be viewed pictorially from Figure 13.2. Linearization starts at the last trait (Trait3) and then we perform it on Trait2 and finally on the BaseClass.

Step 2: In the next step we find all the elements on Trait3 path that are also present on the Trait2 path. For the example program, these are BaseClass, Scala.AnyRef and Scala.Any. We eliminate these elements from the path of Trait3 as can be seen from Figure 13.3.

Step 3: The above mentioned procedure is repeated for all the super types of the DerivedClass and the remaining elements (that are not eliminated) can be seen from Figure 13.4.

Step 4: Finally the linearized traversal from the DerivedClass can be seen as dashed line from Figure

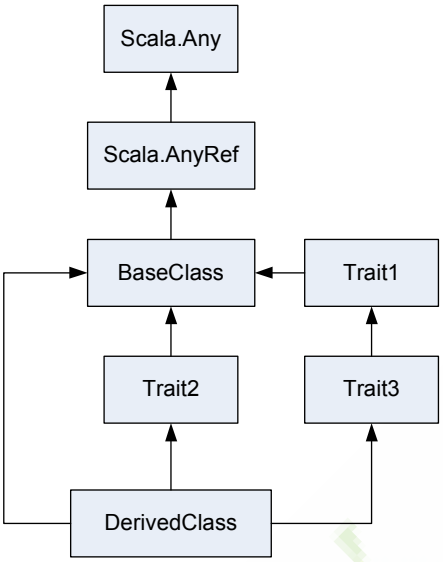


Figure 13.1: Inheritance structure for the example program.

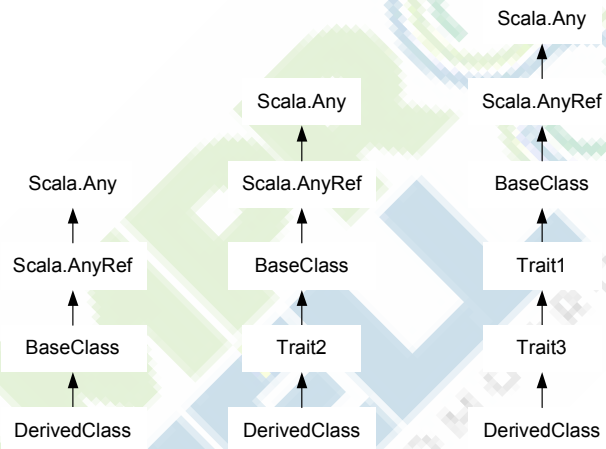


Figure 13.2: Linearize step 1.

13.5.

13.3 Partial function

Partial functions are partial implementations and are also termed as *unary* functions. If a function is partial function then it is not evaluated for every possible value of input parameters. In Scala, partial functions can be defined by using `case` statement. Furthermore the method *orElse* allows chaining other partial function(s).

```
trait PartialFunction[-A, +B] extends (A) => B
```

The `(A) => B` can be interpreted as a function that transforms (a functional mapping) type A input to type B result. The use of partial functions is illustrated for the implementation of square root as provided in Listing 13.3.

```
val squareRoot: PartialFunction[Double, Double] = {
```

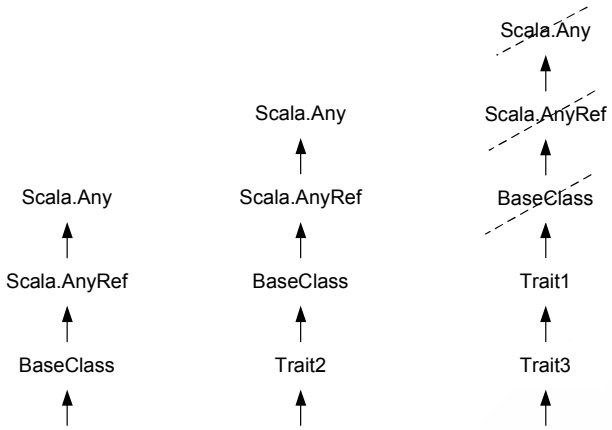


Figure 13.3: Linearize step 2.

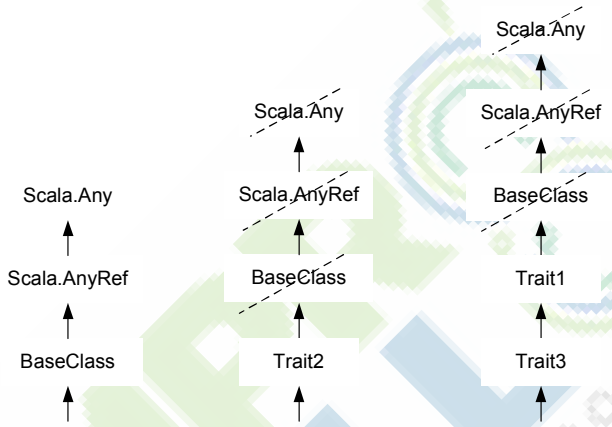


Figure 13.4: Linearize step 3.

```
    case d: Double if d > 0 => Math.sqrt(d)
}
val list: List[Double] = List(4, 16, 25, -9)
val result = list.collect(squareRoot)
// OR
val result = list collect squareRoot

// The output is shown below
result: List[Double] = List(2.0, 4.0, 5.0)
```

Listing 13.2: Implementing square-root as partial function.

Same result can be obtained using the map method.

```
// Doing the same with the map method
val list: List[Double] = List(4, 16, 25, -9)
val result = list.map(Math.sqrt)
result: List[Double] = List(2.0, 4.0, 5.0, NaN)
```

Listing 13.3: Implementing square-root as partial function.

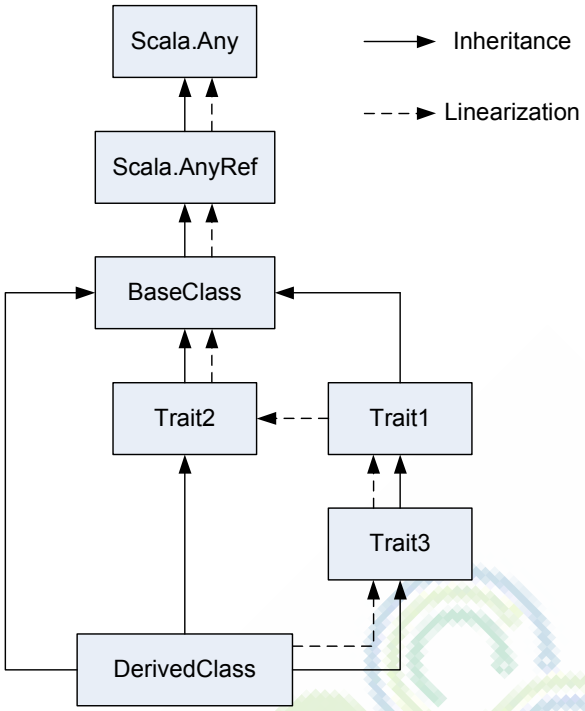


Figure 13.5: Linearize step 4.

13.4 Advanced Parameterization

To this point, different types of parameterization techniques have been discussed. Now we are going to illustrate how to use modules as parameters. In this process, the use of traits is also elaborated. Specifically, the IO interface for the module is defined as a trait. This is illustrated in Listing 13.4. The derived class `Top` accepts a parameter of module type. Two user modules, namely `Add` and `Sub` have also been defined, which can be passed as parameters to the class Top. Furthermore, you can notice the use of multiple IO constructs in user defined Add and Sub modules, illustrating the multi-IO interface based connectivity.

```
import chisel3._
import chisel3.util._
import chisel3.experimental.{BaseModule}

// Define IO interface as a Trait
trait ModuleIO {
    def in1: UInt
    def in2: UInt
    def out: UInt
}

class Add extends RawModule with ModuleIO {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 + in2
}
```

```
class Sub extends RawModule with ModuleIO {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 - in2
}

class Top [T <: BaseModule with ModuleIO] (genT: => T) extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })
    val sub_Module = Module(genT)
    io.out := sub_Module.out
    sub_Module.in1 := io.in1
    sub_Module.in2 := io.in2
}

// Generate verilog for two modules, one for addition, second for subtraction
println((new chisel3.stage.ChiselStage).emitVerilog(new Top(new Add)))
println((new chisel3.stage.ChiselStage).emitVerilog(new Top(new Sub)))
```

Listing 13.4: Module as parameter example code.

13.5 Excersice

Exercise 1: Write an `apply` function, which when called adds a complete list that it is called upon and prints out its value.

Exercise 2: Refer to Listing 13.4 and modify it to pass two modules as parameters at the same time and show their output at two different output ports.

Experiment 14

Scala IV

Objective

To become familiar with an advance Scala concept of Implicit parameters and understand their working and usage in the context of configurable hardware generation.

14.1 The Implicit Keyword

The keyword implicit can be used to define implicit function, implicit class and implicit parameters. The implicit keyword makes the class's primary constructor available for implicit conversions when the class is in the scope.

14.1.1 Implicit Function

An implicit function provides extension method and is called automatically when required by the compiler and is in the scope. Listing 14.1 illustrates the use of implicit function. It can be observed that the `map` method is not available for Int datatype and we convert the Int type variable i to a Seq (specifically to a list) before using the map method explicitly. The same conversion is also performed implicitly by defining an implicit method (`any-name`), which is invoked by `i.map`.

```
import scala.language.implicitConversions

class Implicit_Function (i : Int) {
    // Explicit conversion to Seq (List) and then increment
    val i_explicit_seq_inc = Seq(i).map {
        n => n + 1
    }

    val i_inc = i + 2

    // Implicit conversion to Seq (List) and then increment
    val i_implicit_seq_inc = i.map(_ + 3)

    // Implicit function for conversion
    implicit def any_name(i: Int): Seq[Int] = Seq(i)
}

val convert = new Implicit_Function(5)
println("Explicit conversion and increment: " + convert.i_explicit_seq_inc)
println("Simple increment: " +convert.i_inc)
println("Implicit conversion and increment: " +convert.i_implicit_seq_inc)
```

```
// The output at the terminal
Explicit conversion and increment: List(6)
Simple increment: 7
Implicit conversion and increment: List(8)
```

Listing 14.1: Implicit function definition and usage.

14.1.2 Implicit Class

An implicit class can not be defined at the top level and must be defined inside a class, object, or package. Listing 14.2 implements a string modifier using an ordinary class, while the same functionality is implemented using an implicit class in Listing 14.3.

```
class StringModifier(s: String) {
    def increment = s.map(c => ((c + 1).toChar).toUpper)
}

val result = (new StringModifier("hal")).increment
println("hal is modified to: " + result)

// The output at the terminal
hal is modified to: IBM
```

Listing 14.2: String modifier using class.

```
implicit class StringModifier(s: String) {
  def increment = s.map(c => ((c + 1).toChar).toUpper)
}

val result = ("hal").increment
println("hal is modified to: " + result)

// The output at the terminal
HAL is modified to: IBM
```

Listing 14.3: String modifier using implicit class.

14.1.3 Implicit Parameters

An implicit parameter to a method or function is annotated with the keyword `implicit`. Implicit parameter values are passed automatically by the compiler when not provided by the programmer. Compiler looks for implicit parameter values in its implicit scope.

The implicit parameter is also used in a curried argument list. We will not go into what currying is but in Scala, we can have more than one argument lists, which is known as *currying* function as illustrated in the following listing.

```
def add(a:Int)(b:Int):Int = a + b
val sum = add(1)(2)// sum : Int = 3
```

Listing 14.4: Curried method exmaple

Here we have an add function with a curried argument list. If we don't pass all the argument list, Scala compiler will give an error. Next we use implicit keyword as follows

```
def add(a:Int)(implicit b:Int):Int = a + b
```

Listing 14.5: Curried method with implicit parameter

Now when we call this method with an incomplete argument list, before giving an error Scala compiler will look if an implicit parameter is available. And if it can find an implicit parameter of appropriate type then it will automatically pass that to the argument list as elaborated below.

```
implicit val x:Int = 2
val sum = add(1) //sum: Int = 3
```

Listing 14.6: Method call with implicit parameter.

Now we can call the same function in two ways add(a)(b) or add(a). This is just a simple example when the complexity of the code increase the use of implicit parameters can also become confusing. So there are some rules, listed below, which we need to keep in mind while using implicit parameters.

1. A method or constructor can only have one implicit parameter list
2. It must be the last parameter list in the set of parameters
3. In case several arguments match the implicit parameter's type, the most specific is chosen using the rules of static overloading resolution

Listing 14.7 illustrates the use of implicit parameters list. If we define both implicit parameters of same type, then it will result in an error, as the compiler will not be able to resolve the assignment of implicit parameters. When a `class` definition combines the use of type parameterization along with implicit parameters, it is termed as `type class`.

```
class LinearMap {
  def multiplyOffset(x: Int) (implicit weight: Int, offset: Float) =
      (weight * x).toFloat + offset
}

// initial value assignment
implicit val weightage = 3
implicit val offset = 4.0F
val x = 5

val scale = new LinearMap
println("The result with Implicit parameters omitted: " + scale.
   multiplyOffset(x))
println("The result with Implicit parameters passed explicitly: " + scale.
   multiplyOffset(x)(weightage, offset))

// output at the terminal is
The result with Implicit parameters omitted: 19.0
```

```
The result with Implicit parameters passed explicitly: 19.0
```

Listing 14.7: Illustration of implicit parameters.

14.2 Lazy val

Before discussing lazy vals, we first elaborate the behavior of val and def at the time of their definitions. Specifically, `vals` are evaluated only once at the time of their definition. Once a val is evaluated, the same value is used for all future references without reevaluation.

However, when a function or method is defined using the keyword `def`, it is not evaluated at the time of definition. Rather its evaluation is postponed till the first call to the method and afterwards it is evaluated each time whenever called.

Contrary to the above, a `lazy val` is not evaluated at the time of definition. Rather its evaluation is delayed till its use for the first time. Once evaluated, the same value is used for all future references, without reevaluation.

```
// val evaluation illustration
object valApp extends App{
    val x = {
        println("x is initialized,"); 99
    }
    println("before we access 'x'")
    println(x + 1)
}

// Output at the terminal
x is initialized,
before we access 'x'
100
```

Listing 14.8: The `val` evaluation.

```
// lazy val illustration
object lazyvalApp extends App{
    lazy val x = {
        println("x is NOT initialized."); 99
    }
    println("Unless we access 'x',")
    println(x + 1)
    println(x + 2)
}

// Output at the terminal
Unless we access 'x',
x is NOT initialized.
100
101
```

Listing 14.9: The lazy val evaluation.

14.2.1 Non-Strict Collections and View Method

Scala collection operations can be grouped into the following two categories:

- transformation operations (also termed as transformers) that return another collection (e.g. filter, map, zip),
- reduction operations that result in a single value (e.g. isEmpty, foldLeft, find).

Following are two ways to implement transformers.

- *strict*: a new collection with all elements is constructed due to transformation
- *lazy*: constructs only a representative (proxy) for resulting collection, while the elements are constructed on demand

By default Scala collections are strict when implementing transformers with the exception of Stream collection. Stream implements all its transformer methods lazily, but has been deprecated in Scala 2.13.1 in favor of LazyList.

Every strict collection, however, can be turned into a lazy one and vice versa, using views. The view is a special kind of collection representing some base collection, but implements all its transformers lazily. Going from a collection to its view, requires invocation of view method on the collection. Converting a collection to its view, can provide performance improvement. For instance, Listing 14.10 illustrates the invocation of map method multiple times on the vector collection. The expression uVec map (_ + 1) constructs a new second vector which is then transformed to a third vector by the call to map (_ * 2).

```
val uVec = Vector(1 to 10: _*)

// variant of expression uVec.map(_ + 1).map(_ * 2)
val uVecMapped = uVec map(_ + 1) map (_ * 2)  // syntactic sugar

println(uVec)
println(uVecMapped)

// Output at the terminal
Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Listing 14.10: Eample vector map.

For performance improvement, we can avoid evaluation of intermediate results. This can be achieved by first converting the vector to its equivalent view collection by invoking the view method. All transformations (or transformer methods) are then applied to the resulting view collection. Finally we force the view collection to the vector as illustrated in Listing 14.11.

```
val uVec = Vector(1 to 10: _*)

// view method and resulting view collection
val uVecView = uVec.view
val iView = uVecView map (_ + 1) map (_ * 2) filter (_ > 0)
val uVecMapped = iView.force

println(iView)
println(uVecMapped)

// Output at the terminal
SeqViewMMF(...)
Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Listing 14.11: Illustration of view collection.

14.3 Exercises

Exercise 1: Implicit can be used in different contexts. Explore and implement the use of implicit for type conversion.

Exercise 2: Write a function which has two inputs of any type and then implicitly converts these inputs of any type to String type and print the addition of these strings.

Appendix A

FIRRTL

A.1 Tools Invocation

- The digital circuit is described in Chisel (**Example.scala**)
- Scala compiler compiles **Example.scala**, together with the Chisel and Scala libraries, and generates **Example.class** (a Java class)
- The **Example.class** is compiled by Chisel Driver to generate **Example.fir**

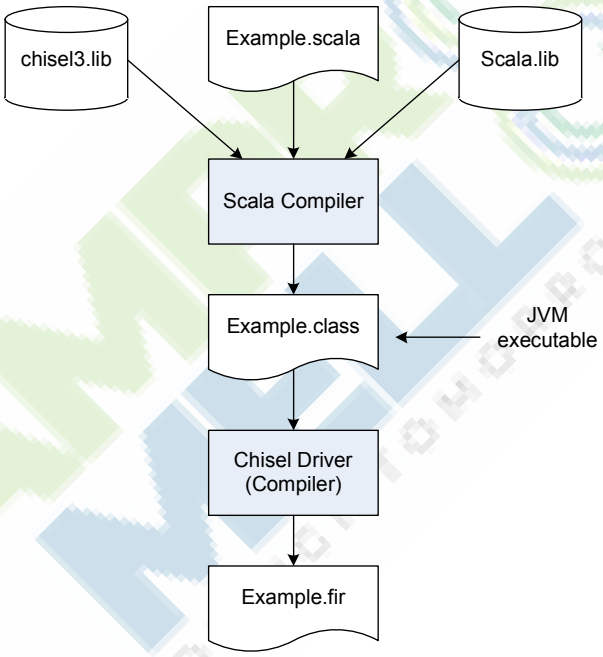


Figure A.1: The tools flow.

A.2 What is FIRRTL

- FIRRTL is a Flexible Intermediate Representation (IR) for RTL (the circuit)
- It is the file format emitted by the Chisel compiler (.fir file)
- The *.fir file is the input to FIRRTL compiler (written in Scala)
- FIRRTL compiler passes the circuit through a series of circuit-level transformations
- A FIRRTL transform can be applied at one of three levels, Hi/Mid/Low
- Transforms can be standalone or can take annotations as input

- Annotations are used to pass information to FIRRTL compiler when applying transforms
- Chisel driver automatically serialize the annotations as json snippet (.json file)
- JSON (JavaScript Object Notation) is a syntax for storing and exchanging data

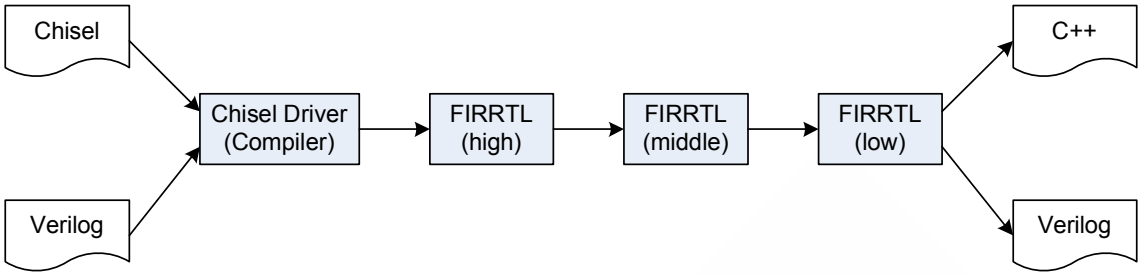


Figure A.2: The FIRRTL stages.

Bibliography

[1] ARM, "Ahb lite spec." http://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf, 2006.

[2] C. E. Laforest, Z. Li, T. O'rourke, M. G. Liu, and J. G. Steffan, "Composing multi-ported memories on fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–23, 2014.

[3] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Incorporation, 2016.

[4] M. Schoeberl, *Digital Design with Chisel*. Kindle Direct Publishing, 2019.

[5] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume ii: Privileged architecture version 1.9," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*, 2016.

[6] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*, 2016.