# Parameterization

Muhammad Tahir

Lecture 5

## Contents

## Parameterization in Chisel

- Powerful feature allowing parameterized hardware generation

## Parameterization in Chisel

- Powerful feature allowing parameterized hardware generation
- Parameterization using bitwidth as parameter

## Parameterization in Chisel

- Powerful feature allowing parameterized hardware generation
- Parameterization using bitwidth as parameter
- Define functions with type parameters

## Parameterization in Chisel

- Powerful feature allowing parameterized hardware generation
- Parameterization using bitwidth as parameter
- Define functions with type parameters
- Bundle parameterization

## Parameterization in Chisel

- Powerful feature allowing parameterized hardware generation
- Parameterization using bitwidth as parameter
- Define functions with type parameters
- Bundle parameterization
- Define modules with type parameters

## Parameterization in Chisel

- Powerful feature allowing parameterized hardware generation
- Parameterization using bitwidth as parameter
- Define functions with type parameters
- Bundle parameterization
- Define modules with type parameters
- Advanced parameterization

## Parameterization: Counter Example Revisited

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

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

    // 'genCounter' with counter size 'n'
    def genCounter(n: Int, max: UInt) = {
        val count = RegInit(0.U(n.W))

        when(count === max) {
            count := 0.U
        }.otherwise {
            count := count + 1.U
        }
        count
    }
```

## Parameterization: Counter Example Revisited Cont'd

```
// genCounter instantiation
val counter1 = genCounter(size, maxValue)
val counter2 = genCounter(size + size, maxValue)
io.result := counter1(size-1) & counter2(0)
}

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

## Type Parameters

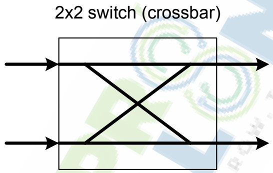
- Generic classes in Scala take a type as a parameter
- A type parameter is specified using square brackets [ ]
- Type parameterization is also termed as *generics*

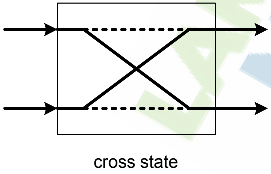
```
class uStack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def pop(): A = {
    val elementTop = elements.head
    elements = elements.tail
    elementTop
  }
}

val stackObj = new uStack[String]
stackObj.push("Hello")
stackObj.push("Trainee")
println(stackObj.pop) // will print Trainee
println(stackObj.pop) // will print Hello
```

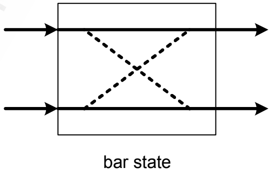
## Functions with Type Parameters

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

class Multiply_Acc(gen: UInt) extends Module {
    val io = IO(new Bundle {
        val out = Output(gen)
        val in1 = Input(gen)
        val in2 = Input(gen)
        val in3 = Input(gen)
    })

    io.out := multiply_add(io.in1, io.in2, io.in3)

    def multiply_add[T <: UInt](in_1: T, in_2: T, in_3: T) =
       {
        in_1 * in_2 + in_3
    }
}

println(chisel3.Driver.emitVerilog(new Multiply_Acc(UInt(2.W
    ))))
```

## Network Switch







## Parameterized Module: 2x2 Switch

```
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))))
```

## Defining Parametrized Bundles

- Define a parameterized class extending the base class `Bundle`
- This class is used to instantiate the IO interface for the module
- Parameters must be passed while creating an IO module
- These parameters themselves might have been derived from the module class
- Different parameterized fields in a bundle can be defined by invoking cloneType on the parameter.

## Parametrized Bundles: Illustration

```
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)))
```

## 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 is to create a custom `cloneType` method in the parameterized bundle

## cloneType Example

```
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]
}
```

## cloneType Example

```
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)))
```

## Class As 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)
```

## Class As Parameter Cont'd

```
  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)))
```

## Function As Parameter

Class with *function parameters*

```
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)
}
```

## Function As Parameter Cont'd

Implementing ADD operation

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

Implementing AND operation

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

## Variance in Scala

- Recall `class` uStack[A]
  - A – is called `Type Parameter`
  - uStack[Int], uStack[String] – are known as `Parameterized Types`
- Variance in Scala defines inheritance relationship among these `Parameterized Types`

## Types of Variance

- Covariance
  - If ‘S’ is subtype of ‘T’ then Set[S] is a subtype of Set[T]
  - Covariance is defined by prefixing Type Parameter with +, e.g. Set[+T], List[+T]
- Contravariance
  - If ‘S’ is subtype of ‘T’ then List[T] is a subtype of List[S]
  - Contravariance is defined by prefixing Type Parameter with –, e.g. Set[-T], List[-T]
- Invariance
  - If ‘S’ is subtype of ‘T’ then List[S] and List[T] do not have any inheritance relationship i.e. they are unrelated (the default variance relationship).

## Reading List I

- Read the relevant sections from Chapter 10 of [Schoeberl, 2019]
- For type parameters and associated variance read Sections 19.3 to 19.6 of [Odersky et al., 2016]

## References

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

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