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.

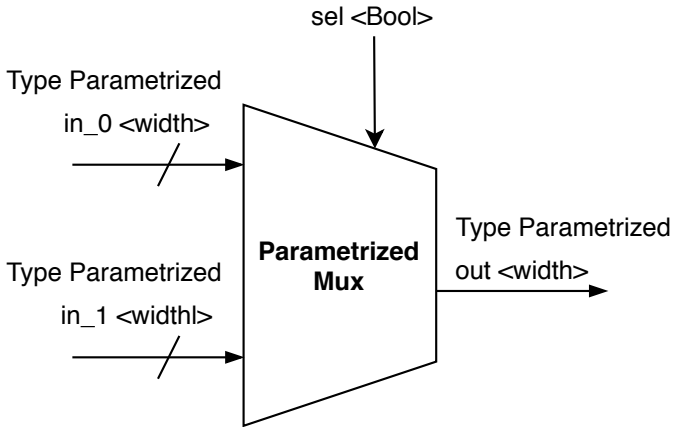

Figure 5.1: Parameterized Mux.

.

```
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.

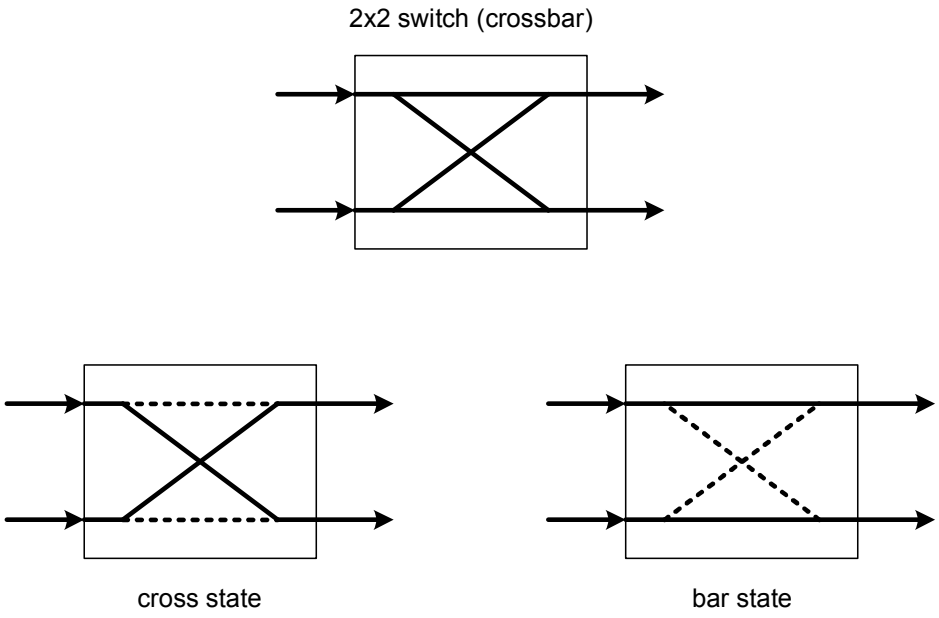

Figure 5.2: Network switch (2x2).

```
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 Listing 5.1 and make an ALU using `when`,`.elsewhen`,`.otherwise` statement.

Exercise 2: Refer to the Listing 5.2, Type parametrize the Mux using bundles only.

Exercise 3: 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 4: Refer to the Listing 5.5. What happens if we don't use clone type? 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 Adder module, the width of the Adder inputs and outputs should be parameterized. You have to use these ports names in your code: io.in0, io.in1, and

io.sum.

```
package Lab5
import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec,Driver,PeekPokeTester}

class Adder(Width:Int) extends Module {
  require(Width >= 0)
// your code begin from here

// your code end here
}
```

Listing 5.9: Skeleton code for Adder.

Task 2: Write chisel code using bundle parameterization to receive and transmit data packets in the Router class. Make one or more separate classes for data packets then use them in Router class. Each data packet must have an address field and data field. Only the data field is type parameterized while the address field is UInt type of ten bits width. You have to use these ports names in your code: io.in and io.out.

```
package Lab5

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

// your code for Transaction_in class


// your code for Transaction_out class

class Router[T<:Data](gen:T) extends Module{
// your code begin


// your code end
}
```

Listing 5.10: Skeleton code for Router.

Task 3: Make an eMux with inclusive typecasting. Use different types for both inputs and explain your analysis and findings.