An Overview of Chisel3
Introduction

- Constructing Hardware In a Scala Embedded Language
- Compilation Pipeline
  - Chisel3 -> FIRRTL (Flexible Internal Representation for RTL) -> Verilog

- Scala executes its statements sequentially
  - Allow declaring nodes that can be used immediately, but whose input will be set later
Datatypes

- SInt, UInt, Bool
- Examples:

```scala
code
val a = 5.S       // signed decimal 4-bit lit from Scala Int
cval b = “b1010”.U // binary 4-bit lit from string
cval c = true.B    // Bool lit from Scala lit
cval d = 5.U(8.W)  // unsigned decimal 8-bit lit of type UInt
cval e = “h_dead_beef”.U  // unsigned 32-bit lit of type UInt
```

- `.W` is used to cast a Scala Int to a Chisel Width
Combinational Circuits and Wires

- A circuit is represented as a graph of nodes
- Each node is a hardware operator that has \( \geq 0 \) inputs and drives 1 output
- Examples:

```scala
val wire1 = false.B
val wire2 = (a & b) | (~c & d)
val wire3 = wire1 ^ wire2
wire3 = true.B  // not allowed

val myWire = Wire(UInt(8.W))  // allocate a wire of type UInt, width 8
myWire := 255.U
myWire := 0.U              // connect to myWire, last one takes effect
```
Registers

- Retain state until updated
- Reg, RegInit, RegNext
- Examples:

```scala
val r1 = Reg(UInt(4.W)) // reg without initialization
val r2 = RegInit(UInt(), 3.U(8.W)) // initialized to 0 upon reset
val r3 = RegInit(wire1) // type and width inferred
val r4 = RegNext(pcNext, 0.U(32.W)) // output a copy of pcNext
  // delayed by one clock cycle
r3 := next_val // assign to latch new value on next clock
```
Memory

- Chisel has a `Mem` construct indexed by address
- Writes to `Mem` are posedge triggered and reads are either asynchronous or posedge triggered (though current FPGA tech does not support async reads anymore?)

```scala
val rf = Mem(256, UInt(8.W))
rf(42) := 3.U
val data = rf(42)
```
Defining Functions

- We can define a piece of logic using Scala’s functional programming support
- Any value expressed on the last line in a block is considered the return value (this is a Scala thing)
- Examples:

```scala
def clb(a: Bool, b: Bool, c: Bool) = a & b | c
val out = clb(true.B, some_wire, false.B)  // usage

def wrapAround(n: UInt, max: UInt) =
  Mux(n > max, 0.U, n)               // functional module creation
```
Aggregate Types

- **Bundles (structs)**
- Define a class as a subclass of **Bundle**

```scala
class MyBundle extends Bundle {
  val field1 = Bool()
  val field2 = UInt(8.W)
}

val x = Wire(new MyBundle)
x.field1 := true.B
val wire1 = x.field1    // true
```
Aggregate Types cont.

- Vecs (arrays)
- 0 indexed

```scala
val myVec = Wire(Vec(5, SInt(23.W)))
myVec(0) := 3.U
val wire0 = myVec(0) // 3
val reg = RegInit(myVec(4)) // 0

```
Ports

- Interface to hardware components
- Simply any Data object that has directions assigned to its fields

```scala
class ScaleIO extends Bundle {
    val valid = Input(Bool())
    val in    = Input(UInt(32.W))
    val scale = Input(UInt(32.W))
    val out   = Output(UInt(32.W))
}
```
Modules

- Very similar to Verilog modules, helps define hierarchical structure in the circuit
- Defined as a class that inherits from `Module`
- Contains an interface wrapped in an `IO()` function stored in a port named `io`
- Wires together subcircuits in its constructor

```scala
class Mux2 extends Module {  // subclass of Module
  val io = IO(new Bundle {  // io port
    val sel = Input(UInt(1.W))
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val out = Output(UInt(1.W))
  })
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```
• Instantiating modules

```scala
val myMux2 = Module(new Mux2())
myMux2.io.sel := 1.U
myMux2.io.in0 := 1.U
myMux2.io.in1 := 0.U

val r1 = Reg(UInt(1.W))
r1 := myMux2.io.out
```
Modules cont. (Functional Module Creation)

- Useful for making more readable hardware connections that are similar to software expression evaluations
- Creates a Scala object on the module class, and `apply` defines a method for creation of a Mux2 instance

```scala
object Mux2 {
  def apply (sel: UInt, in0: UInt, in1: UInt) = {
    val m = Module(new Mux2)
    m.io.in0 := in0
    m.io.in1 := in1
    m.io.sel := sel
    m.io.out  // return value
  }
}

val r1 = Mux2(1.U, 1.U, 0.U)  // much simpler
```
Conditional Updates

- Can be used to specify when updates to registers will occur
- Update blocks can only contain statements using :=, simple expressions, named wires defined with val

```plaintext
when (c1) { u1 }
 .elsewhen (c2) { u2 }
 .otherwise { ud }

switch (idx) {
    is(v1) { u1 }
    is(v2) { u2 }
}
```
Conditional Updates cont.

- Sequences of conditional updates (last one takes precedence)
- Overlapping subsets of registers in different update blocks
- A register is only affected by conditions in which it appears

```scala
def main() {
  val r = Reg(SInt(2.W)); val s = Reg(SInt(2.W))
  r := 3.S; s := 3.S
  when (c1) {r := 1.S; s := 1.S}
  when (c2) {r := 2.S}
}
```

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
State Elements

- Chisel supports a global clock and reset
- Basic := and Register operators update are positive edge-triggered
- Can use this to create useful counters and pulse generators

```scala
def counter(max: UInt) = {
  val x = Reg(init = 0.U(max.getWidth.W))
  x := Mux(x === max, 0.U, x + 1.U)
  x
}
// Produce pulse every n cycles.
def pulse(n: UInt) = counter(n - 1.U) === 0.U
```
Nifty things

- Print statements
  - Both C style and Scala style
    
    `printf(...)`

- Built-in Muxes
  
  `Mux(s, i1, i0)`
  
  `MuxCase(default, Array(c1 -> a, c2 -> b, ...))`
  
  `MuxLookup(idx, default, Array(0.U -> a, 1.U -> b, ...))`