Chisel Tutorial

https://github.com/ucb-bar/chisel-tutorial

"Chisel is an open-source hardware construction language (..) that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages."

exercises with circuits: `src/main/scala/problems`
associated test harnesses: `src/test/scala/problems`
solutions: `src/main/scala/solutions` **&** `src/test/scala/solutions`

Mux2

```
class Mux2 extends Module {
  val io = IO(new Bundle {
    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)
}
```

Mux4

```
class Mux4 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val in2 = Input(UInt(1.W))
    val in3 = Input(UInt(1.W))
    val sel = Input(UInt(2.W))
    val out = Output(UInt(1.W))
  })

  val m0 = Module(new Mux2())
  m0.io.sel := io.sel(0)
  m0.io.in0 := io.in0
  m0.io.in1 := io.in1

  val m1 = Module(new Mux2())
  m1.io.sel := io.sel(0)
  m1.io.in0 := io.in2
  m1.io.in1 := io.in3

  val m2 = Module(new Mux2())
  m2.io.sel := io.sel(1)
  m2.io.in0 := m0.io.out
  m2.io.in1 := m1.io.out

  io.out := m2.io.out
}
```

Counter

Nota: You can conditionally update a value without a mux by using `when (cond) { foo := bar }

```
class Counter extends Module {
  val io = IO(new Bundle {
    val inc = Input(Bool())
    val amt = Input(UInt(4.W))
    val tot = Output(UInt(8.W))
  })
  io.tot := Counter.counter(255.U, io.inc, io.amt)
}
```

```
object Counter {
  def wrapAround(n: UInt, max: UInt) =
    Mux(n > max, 0.U, n)

  // Modify below ----------
  def counter(max: UInt, en: Bool, amt: UInt): UInt = {
    val x = RegInit(0.U(max.getWidth.W))
    //if doesn't work because "en" is a chisel.bool
    when (en) {x := wrapAround(x + amt, max)}
    x
  }
  // Modify above ----------
}
```

Acumulador

Nota: Necessário Registo para incrementar próprio valor

```scala
class Accumulator extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(1.W))
    val out = Output(UInt(8.W))
  })
  // Implement below ----------
  val x = RegInit(0.U(8.W))
  when(io.in === 1.U){
  x := x + 1.U
  }
  io.out := x

  // Implement above ----------
}
```

ou

```scala
class Accumulator extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(1.W))
    val out = Output(UInt(8.W))
  })
  val accumulator = RegInit(0.U(8.W))
  accumulator := accumulator + io.in
  io.out := accumulator
}
```

Adder

```scala
// 'out' should be the sum of 'in0' and 'in1'
// Adder width should be parametrized
//
class Adder(val w: Int) extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(w.W))
    val in1 = Input(UInt(w.W))
    val out = Output(UInt(w.W))
  })
  io.out := io.in0 + io.in1
}
```

Max2

```scala
val in0 = rnd.nextInt(10)
val in1 = rnd.nextInt(10)
poke(c.io.in0, in0)
poke(c.io.in1, in1)
step(1)
expect(c.io.out, if (in0 > in1) in0 else in1)
```

MaxN

```scala
class MaxNTests(c: MaxN) extends PeekPokeTester(c) {
  val arr = Array.fill(c.n){ 0 }
  for (i <- 0 until 10) {
    var mx = 0
    for (i <- 0 until c.n) {
      arr(i) = rnd.nextInt(1 << c.w)
      poke(c.io.ins(i), arr(i))
      mx = if (arr(i) > mx) arr(i) else mx
    }
    step(1)
    expect(c.io.out, mx)
  }
}
```

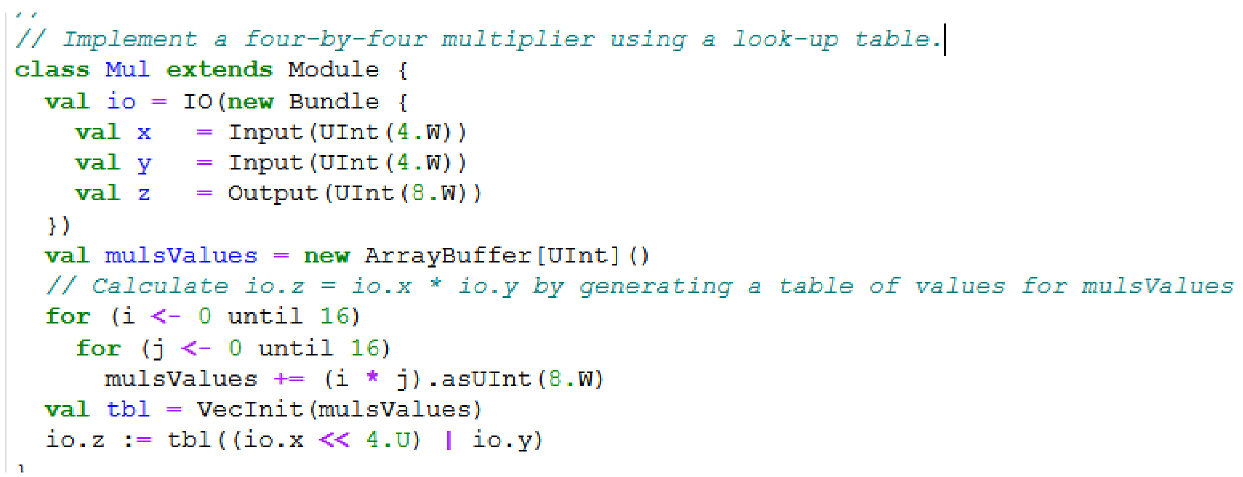
Nota: No “tester”, para criar variável de Width dinâmico (n) usa-se: **(1 << n)**

Mul – Look-up Table

The main building block of combinatorial logic in an FPGA is called a lookup table, but usually abbreviated as *LUT*. This is just a small RAM element that takes 4 or 5 or 6 inputs (depending on which type of FPGA) and uses that to select a bit from memory to be output.

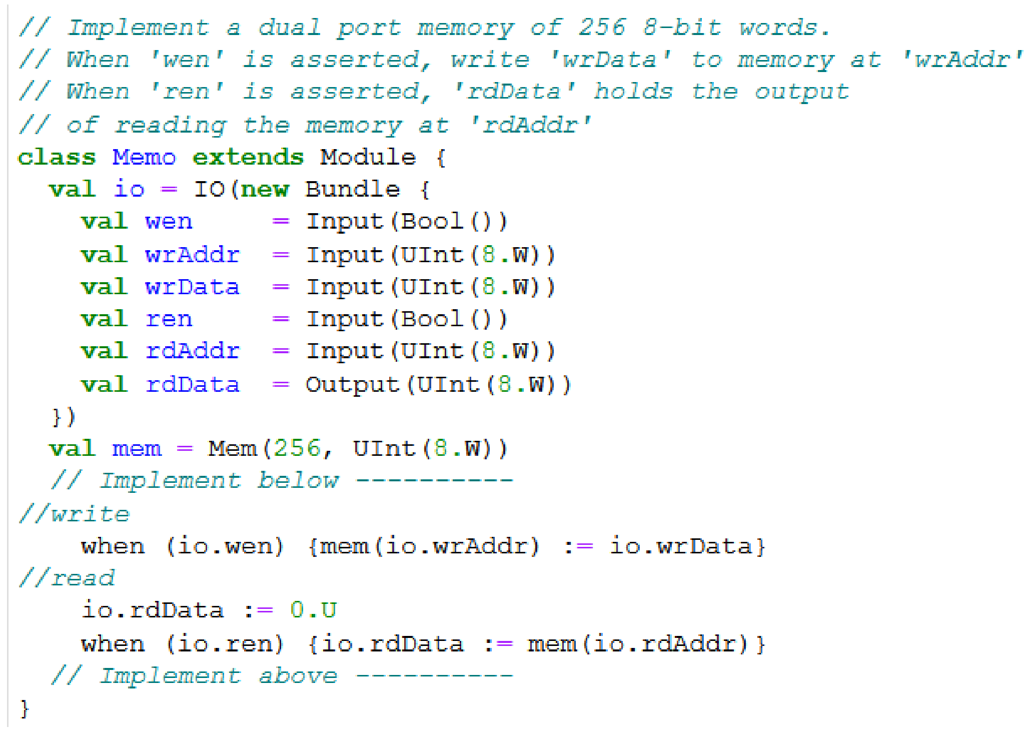
Second, more generally a lookup table is just logic that takes inputs and has a defined output for every combination of those inputs, just like a logic table you'd use for paper-and-pencil design.

In Verilog, a lookup table is usually implemented with a case statement.



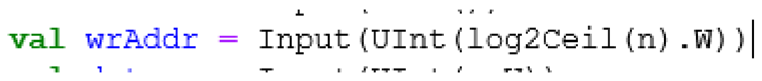
```
// Implement a four-by-four multiplier using a look-up table.
class Mul extends Module {
  val io = IO(new Bundle {
    val x   = Input(UInt(4.W))
    val y   = Input(UInt(4.W))
    val z   = Output(UInt(8.W))
  })
  val mulsValues = new ArrayBuffer[UInt]()
  // Calculate io.z = io.x * io.y by generating a table of values for mulsValues
  for (i <- 0 until 16)
    for (j <- 0 until 16)
      mulsValues += (i * j).asUInt(8.W)
  val tbl = VecInit(mulsValues)
  io.z := tbl((io.x << 4.U) | io.y)
```

MEMORY



```
// Implement a dual port memory of 256 8-bit words.
// When 'wen' is asserted, write 'wrData' to memory at 'wrAddr'
// When 'ren' is asserted, 'rdData' holds the output
// of reading the memory at 'rdAddr'
class Memo extends Module {
  val io = IO(new Bundle {
    val wen     = Input(Bool())
    val wrAddr  = Input(UInt(8.W))
    val wrData  = Input(UInt(8.W))
    val ren     = Input(Bool())
    val rdAddr  = Input(UInt(8.W))
    val rdData  = Output(UInt(8.W))
  })
  val mem = Mem(256, UInt(8.W))
  // Implement below ----------
//write
    when (io.wen) {mem(io.wrAddr) := io.wrData}
//read
    io.rdData := 0.U
    when (io.ren) {io.rdData := mem(io.rdAddr)}
  // Implement above ----------
}
```

DYNAMIC MEMORY SEARCH



```
val wrAddr = Input(UInt(log2Ceil(n).W))
```

“*log2Ceil()*” → ao fazer o logaritmo base 2 (arredondado para cima) de N sabemos quantos bits ou o tamanho que o wrAddr tem de ter para poder endereçar N valores.

```
// This module should be able to write 'data' to
// internal memory at 'wrAddr' if 'isWr' is asserted.
//
// This module should perform sequential search of 'data'
// in internal memory if 'en' was asserted at least 1 clock cycle
//
// While searching 'done' should remain 0,
// 'done' should be asserted when search is complete
//
// If 'data' has been found 'target' should be updated to the
// address of the first occurrence
//
class DynamicMemorySearch(val n: Int, val w: Int) extends Module {
  val io = IO(new Bundle {
    val isWr   = Input(Bool())
    val wrAddr = Input(UInt(log2Ceil(n).W))
    val data   = Input(UInt(w.W))
    val en     = Input(Bool())
    val target = Output(UInt(log2Ceil(n).W))
    val done   = Output(Bool())
  })
  val index  = RegInit(0.U(log2Ceil(n).W))
  // Implement below ----------
  //implementar memória
  val list = Mem(n, UInt(w.W)) //segundo parametro é para definir o tipo
  val memVal = list(index)
  // Implement above ----------
  val done   = !io.en && ((memVal === io.data) || (index === (n-1).asUInt))
  // Implement below ----------
    when (io.isWr){
        list(io.wrAddr) := io.data
    }
    // Implement above ----------
    when (io.en) {
        index := 0.U
      } .elsewhen (done === false.B) {
        index := index + 1.U
      }
  io.done   := done
  io.target := index
}
```

LFSR16

```
// Implement below ----------
val lfsr = RegInit(UInt(16.W), 1.U)
val bit = ((lfsr >> 0)^(lfsr >> 2)^(lfsr >> 3)^(lfsr >> 5))
val nextState = (lfsr >> 1) | (bit << 15);
when(io.inc){lfsr := nextState}
io.out := lfsr
// Implement above ----------
```

Para ficar igual era colocar o Bit como WIRE.

Solução

```
val res = RegInit(1.U(16.W))
when (io.inc) {
  val nxt_res = Cat(res(0)^res(2)^res(3)^res(5), res(15,1))
  res := nxt_res
}
io.out := res
```

GREATEST COMMON DIVIDER

```
// Problem:
// Implement a GCD circuit (the greatest common divisor of two numbers).
// Input numbers are bundled as 'RealGCDInput' and communicated using the Decoupled handshake protocol
//
class RealGCDInput extends Bundle {
  val a = UInt(16.W)
  val b = UInt(16.W)
}

class RealGCD extends Module {
  val io  = IO(new Bundle {
    val in  = DeqIO(new RealGCDInput())
    val out = Output(Valid(UInt(16.W)))
  })
  val x = Reg(UInt())
  val y = Reg(UInt())
  val p = RegInit(false.B)
  val i = RegInit(1.U(16.W))
  val gcd = RegInit(0.U(16.W))

  io.in.ready := !p
  when (io.in.valid && !p) {
    x := io.in.bits.a
    y := io.in.bits.b
    p := true.B
  }
  when (p) {
    when (x < y)  { x := y; y := x; }
    when (x % i === 0.U && y % i === 0.U)   { gcd := i}
    when (i < (y + 1.U)) {i := i +% 1.U}
      .otherwise {i := 1.U; gcd := 0.U}
  }
  io.out.bits  := gcd
  io.out.valid := ((y + 1.U) === i) && p
  when (io.out.valid) {
    p := false.B
  }
}
```

Exercício complicado.

Não consegui á primeira tive de orientar um pouco pela solução para construir segundo o Protocolo Decoupled.

Para tal, no cálculo são usados registos em vez de combinação lógica instantânea. Necessário atribuir ao circuito valores a **in.ready, out.bits** e **out.valid.**

Para evitar andar com as flags de trás para a frente a definir o estado cria-se o registo bool “*p*”

Muitas complicações em termos de erros de lógica e de arritmia e ethe cast de variáveis

VENDING MACHINE – when() & switch()

```
io.valid := 0.U
when (state === 0.U) {
  when (io.nickel && !io.dime) {
    state  := 1.U
  }.elsewhen(!io.nickel && io.dime) {
    state  := 2.U
  }
}
when (state === 1.U) {
  when (io.nickel && !io.dime) {
    state  := 2.U
  }.elsewhen(!io.nickel && io.dime) {
    state  := 3.U
  }
}
when (state === 2.U) {
  when (io.nickel && !io.dime) {
    state  := 3.U
  }.elsewhen(!io.nickel && io.dime) {
    state  := 4.U
  }
}
when (state === 3.U) {
    state  := 4.U
  }
when (state === 4.U) {
    state  := 0.U
```

```
switch (state) {
  is (sIdle) {
    when (io.nickel) { state := s5 }
    when (io.dime) { state := s10 }
  }
  // Implement below ----------
  is (s5) {
    when (io.nickel) { state := s10 }
    when (io.dime) { state := s15 }
  }
  is (s10) {
    when (io.nickel) { state := s15 }
    when (io.dime) { state := sOk }
  }
  is (s15) {
    state := sOk
  }
  is (sOk) {
   state := sIdle
  }
```

VEC SHIFT REGISTER

De facto mais fácil fazer o registo ao usarmos um VEC como REG

```
class VecShiftRegister extends Module {
  val io = IO(new Bundle {
    val ins   = Input(Vec(4, UInt(4.W)))
    val load  = Input(Bool())
    val shift = Input(Bool())
    val out   = Output(UInt(4.W))
  })
  // Implement below ----------
  io.out := 0.U
  val register = Reg(Vec(4, UInt(4.W)))
  when(io.load){
       register(0) := io.ins(0)
       register(1) := io.ins(1)
       register(2) := io.ins(2)
       register(3) := io.ins(3)
  }
  .elsewhen(io.shift){
       register(0) := io.ins(0)
       register(1) := register(0)
       register(2) := register(1)
       register(3) := register(2)
  }
  io.out := register(3)
  // Implement above ----------
}
```

PARAMETRIZED SHIFT REGISTER

Para criar e inicializar um VEC de registos a 0.U com tamanha e tipo variável:

```
val my_reg = RegInit(VecInit(Seq.fill(n){0.U(w.W)}))
```

```
class VecShiftRegisterParam(val n: Int, val w: Int) extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(w.W))
    val out = Output(UInt(w.W))
  })
  // Implement below ----------
  io.out := 0.U
  val my_reg = RegInit(VecInit(Seq.fill(n){0.U(w.W)}))
  my_reg(0) := io.in
  for( i <- 1 until n){
      my_reg(i) := my_reg(i - 1)
  }
  io.out := my_reg(n-1)
}
```

`chisel3.Data` is the base class for Chisel hardware types. `UInt`, `SInt`, `Vec`, `Bundle` o problema resolvido de outra maneira era mais fácil mas eles decidiram aplicar tipos a tamanhos variáveis, abstração de classes e o uso de objetos como filtros.

```scala
abstract class Filter[T <: Data](dtype: T) extends Module {
  val io = IO(new Bundle {
    val in = Input(Valid(dtype))
    val out = Output(Valid(dtype))
  })
}

class PredicateFilter[T <: Data](dtype: T, f: T => Bool) extends Filter(dtype) {
  io.out.valid := io.in.valid && f(io.in.bits)
  io.out.bits  := io.in.bits
}

object SingleFilter {
  def apply[T <: UInt](dtype: T) =
    // Change function argument of Predicate filter below ----------
    Module(new PredicateFilter(dtype, (x: T) => x <= 9.U))
    // Change function argument of Predicate filter above ----------
}

object EvenFilter {
  def apply[T <: UInt](dtype: T) =
    // Change function argument of Predicate filter below ----------
    Module(new PredicateFilter(dtype, (x: T) => (x%2.U === 0.U)))
    // Change function argument of Predicate filter above ----------
}

class SingleEvenFilter[T <: UInt](dtype: T) extends Filter(dtype) {
  // Implement composition below ----------
    val single = SingleFilter(dtype)
    val even = EvenFilter(dtype)
    single.io.in := io.in
    even.io.in := single.io.out
    io.out := even.io.out
  // Implement composition above ----------
}
```

EXEMPLOS IMPORTANTES:

FullAdder: implementa 1 full adder e a sua lógica

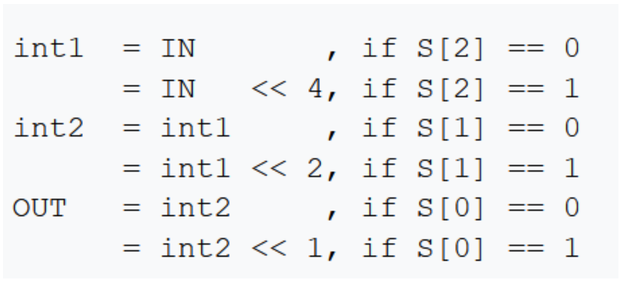
Adder: an Array of FullAdders tamanho dinámico → A n-bit adder tamanho dinámico with carry in and carry out. Contudo não implementa a lógica, chamando o modulo anterior **FullAdder**

Byte Selector: Numa word ou algo selecionar um byte consoante o offset

HiLoMultiplier: A 16*16-bit multiplier with separate high and low product outputs. (32 bite separados em 16 high e 16 low)

Life: Alta javardice, mas um modelo que ele tentou criar entre células e vida com um conjunto de restrições dele

Logshifter: A common usage of a barrel/ Logarithmic shifter is in the hardware implementation of floating-point arithmetic. (floating-point add or subtract operation)



```
int1  = IN      , if S[2] == 0
      = IN   << 4, if S[2] == 1
int2  = int1    , if S[1] == 0
      = int1 << 2, if S[1] == 1
OUT   = int2    , if S[0] == 0
      = int2 << 1, if S[0] == 1
```

Risc: implementação de uma arquitetura RISC.
Router: implementação de um Router.

SimpleALU: simples implementação de uma ALU (Arithmetic logic unit), com 8 instruções

Stack: Implementação de uma Stack 32bits, tamanho N