Introduction to Chisel

Martin Schoeberl

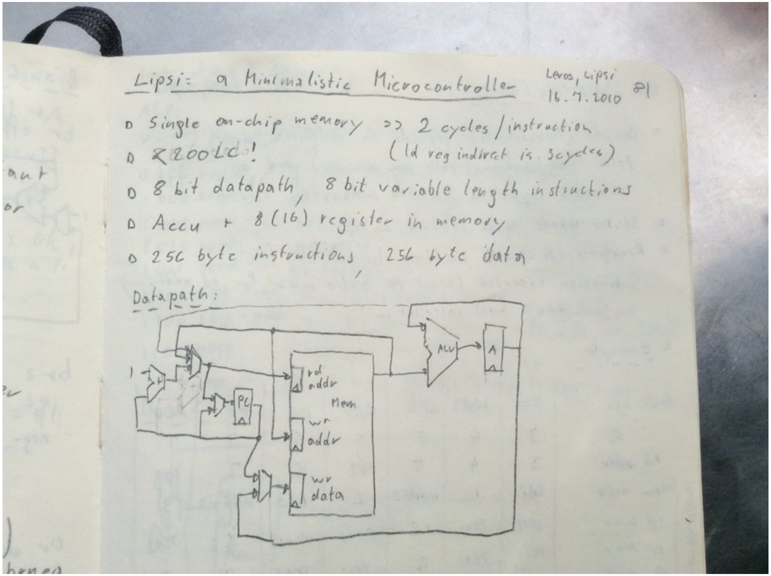
Technical University of Denmark
Embedded Systems Engineering

September 9, 2025

Motivating Example: Lipsi: Probably the Smallest Processor in the World

- Tiny processor
- Simple instruction set
- Shall be small
  - Around 200 logic cells, one FPGA memory block
- Hardware described in Chisel
- Available at `https://github.com/schoeberl/lipsi`
- Usage
  - Utility processor for small stuff
  - In teaching introduction to computer architecture
- The design took place on the island of Lipsi

The Design of Lipsi on Lipsi



Lipsi: a Minimalistic Microcontroller

Leros, Lipsi
16.7.2010 81

- Single on-chip memory >> 2 cycles/instruction
- ≤ 200 LC! (ld reg indirect is 3 cycles)
- 8 bit datapath, 8 bit variable length instructions
- Accu + 8 (16) register in memory
- 256 byte instructions, 256 byte data

Datapath:

1
PC
rd addr
Mem
wr addr
wr data
ALU
A

Lipsi Implementation

- Hardware described in Chisel
- Tester in Chisel/Scala
- Assembler in Scala
  - Core case statement about 20 lines
- Reference design of Lipsi as a software simulator in Scala
- Testing:
  - Self-testing assembler programs
  - Comparing hardware with a software simulator
- All in a single programming language!
- All in a single program
- How much work is this?

Chisel is Productive

- All coded and tested in less than 14 hours!
- The hardware in Chisel
- Assembler in Scala
- Some assembler programs (blinking LED)
- Simulation in Scala
- Two testers
- BUT, this does not include the design (done on paper)

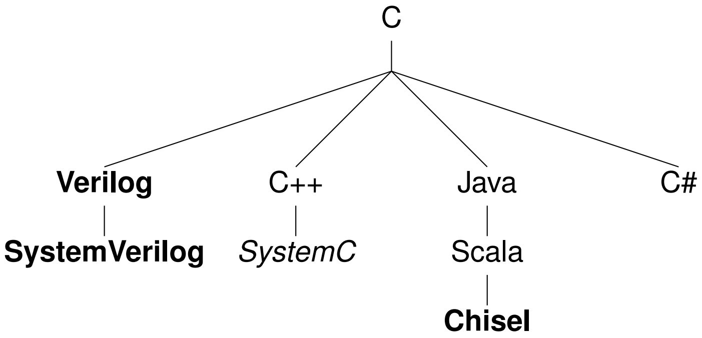
Motivating Example: Lipsi, a Tiny Processor

- Show in IntelliJ

Chisel

- A hardware *construction* language
  - Constructing Hardware in a Scala Embedded Language
  - If it compiles, it is synthesizable hardware
  - Say goodbye to your unintended latches
- Chisel is not a high-level synthesis language
- Single source for two targets
  - Cycle accurate simulation (testing)
  - Verilog for synthesis
- Embedded in Scala
  - Full power of Scala available
  - We use Scala to write the generators
- Developed at UC Berkeley

The C Language Family



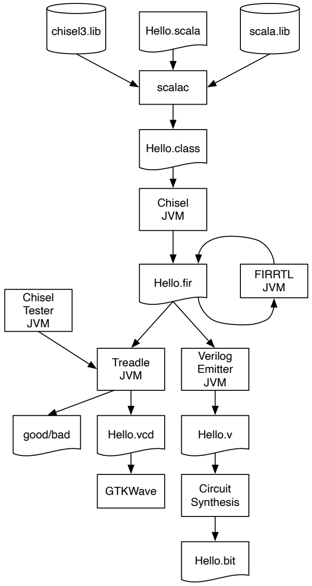
Other Language Families

$$\begin{tikzcd}
\text{Algol} \arrow[d, dash] & \text{Python} \arrow[d, dash] \\
\text{Ada} \arrow[d, dash] & \textbf{MyHDL} \\
\textbf{VHDL} &
\end{tikzcd}$$

A Small Language

- Chisel is a *small* language
- On purpose
- Not many constructs to remember
- The Chisel Cheatsheet fits on two pages
- The power comes with Scala for circuit generators
- With Scala, Chisel can grow with you

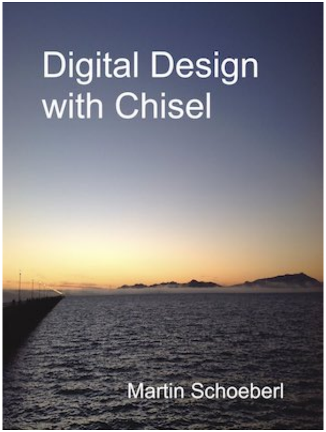
Tool Flow for Chisel Defined Hardware



Chisel is a Hardware Construction Language

- The code looks much like Java code
- But it is *not* a program in the usual sense
- It represents a circuit
- The “program” constructs the circuit
- All statements are “executed” in parallel
- Statement order has *mostly* no meaning

A Chisel Book



- Available in open access (as PDF)
  - Optimized for reading on a tablet (size, hyperlinks)
- Amazon can do the printout

Chisel and Scala

- Chisel is a library written in Scala
  - Import the library with `import chisel3._`
- Chisel code is Scala code
- When it is run is *generates* hardware
  - Verilog for synthesis and simulation
- Chisel is an embedded domain-specific language
- Two languages in one can be a little bit confusing

Chisel in Scala

- Chisel components are Scala classes
- Chisel code is in the constructor
- Executed at object creation time
- Builds the network of hardware objects
- Testers are written in Scala to drive the tests

Signal Types

- All types in hardware are a collection of bits
- The base type in Chisel is `Bits`
- `UInt` represents an unsigned integer
- `SInt` represents a signed integer (in two's complement)

```
Bits(8.W)
UInt(8.W)
SInt(10.W)
```

Number of Bits: n.W

- A collection of bits has a *width*
- The width is the number of bits
- Is written as number followed by `.W`
- Following example shows the width of `n`

```
n.W
Bits(n.W)
```

Constants

- Constants can represent signed or unsigned numbers
- We use `.U` and `.S` to distinguish

```
0.U  // defines a UInt constant of 0
-3.S // defines a SInt constant of -3
```

- Constants can also be specified with a width

```
3.U(4.W) // An 4-bit constant of 3
```

Hexadecimal and Binary Representation

- We can specify constants with a different base
- May come handy sometimes

```
"hff".U        // hexadecimal representation of
   255
"o377".U       // octal representation of 255
"b1111_1111".U // binary representation of 255
```

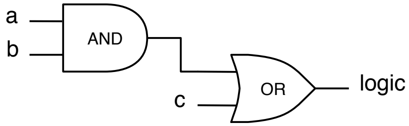
Boolean Values

- Type for logical values
- Can be `true` or `false`
- Almost exchangeable with `UInt(1.W)`
- Sometimes a signal, such as `valid`, may be better represented by a Boolean type

```
Bool()
true.B
false.B
```

Combinational Circuits

- Chisel uses Boolean operators, similar to C or Java
- & is the AND operator and `|` is the OR operator
- The following code is the same as the schematics
- `val logic` gives the circuit/expression the name `logic`
- That name can be used in following expressions



```
val logic = (a & b) | c
```

Combinational Circuits

- Simple expressions represent a circuit tree
- Arbitrary directed acyclic graphs need named subexpressions
- Using Scala's `val` keyword for variables that don't change
- Referenced multiple times

```
val cond = a & b
val result = (cond & selA) | (!cond & selB)
```

Standard Logic Operations

```
val and = a & b // bitwise and
val or = a | b  // bitwise or
val xor = a ^ b // bitwise xor
val not = ~a    // bitwise negation
```

- Note that we do not need to define the width of the values
- Note also that this is *hardware*
- All expressions are evaluated in parallel
- Order does not matter

Arithmetic Operations

- Same as in Java or C
- The width of the result is automatically computed
- E.g., the width of the multiplication is the sum of the width of `a` and the width of `b`

```
val add = a + b // addition
val sub = a - b // subtraction
val neg = -a    // negate
val mul = a * b // multiplication
val div = a / b // division
val mod = a % b // modulo operation
```

Comparison

- The usual operations (as in Java or C)
  - Unusual equal and unequal operator symbols
  - To keep the original Sala operators usable for references
- Operands are `UInt` and `SInt`
- Operands can be `Bool` for equal and unequal
- Result is `Bool`

```
>, >=, <, <=
===, =/=
```

Boolean Logical Operations

- Operands and result are `Bool`
- Logical NOT, AND, and OR

```
val notX = !x
val bothTrue = a && b
val orVal = x || y
```

Chisel Defined Hardware Operators

Operator	Description	Data types
`* / %`	multiplication, division, modulus	UInt, SInt
`+ -`	addition, subtraction	UInt, SInt
`=== =/=`	equal, not equal	UInt, SInt, returns Bool
`> >= < <=`	comparison	UInt, SInt, returns Bool
`<< >>`	shift left, shift right (sign extend on SInt)	UInt, SInt
`~`	NOT	UInt, SInt, Bool
`& | ^`	AND, OR, XOR	UInt, SInt, Bool
`!`	logical NOT	Bool
`&& ||`	logical AND, OR	Bool

Wires

- A signal (or wire) can be first defined
- And later assigned an expression with `:=`

```
val w = Wire(UInt())

w := a & b
```

Connections

- Connections with the `:=` assignment
- When *reassigning* a value to a wire, port, or register

```
adder.io.a := ina
adder.io.b := inb
```

Subfields and Concatenation

A single bit can be extracted as follows:

```
val sign = x(31)
```

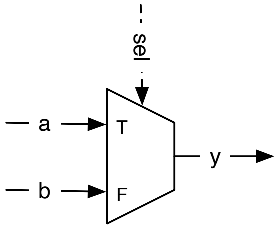
A subfield can be extracted from end to start position:

```
val lowByte = largeWord(7, 0)
```

Bit fields are concatenated with the `##` operator:

```
val word = highByte ## lowByte
```

A Multiplexer



- A Multiplexer selects between alternatives
- So common that Chisel provides a construct for it
- Selects `a` when `sel` is `true.B` otherwise `b`

```
val result = Mux(sel, a, b)
```

Combinational Circuit with Conditional Update

- Value first needs to be wrapped into a `Wire`
- Updates with the Chisel update operation `:=`
- With `when` we can express a conditional update
- The condition is an expression with a Boolean result
- The resulting circuit is a multiplexer
- The rule is that the last enabled assignment counts
  - Here the order of statements has a meaning

```
val enoughMoney = Wire(Bool())

enoughMoney := false.B
when (coinSum >= price) {
  enoughMoney := true.B
}
```

The “Else” Branch

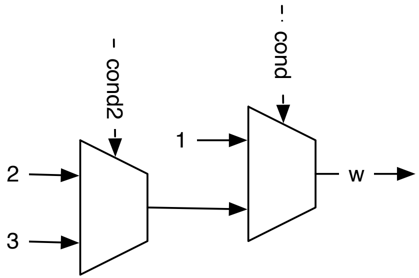
- We can express a form of “else”
- Note the `.` in `.otherwise`

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .otherwise {
  w := 2.U
}
```

A Chain of Conditions

- To test for different conditions
- Select with a priority order
- The first that is true counts
- The hardware is a chain of multiplexers

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .elsewhen (cond2) {
  w := 2.U
} .otherwise {
  w := 3.U
}
```



Default Assignment

- Practical for complex expressions
- Forgetting to assign a value at all conditions
  - Would describe a latch
  - Runtime error in Chisel
- Assign a default value is a good practice

```
val w = WireDefault(0.U)

when (cond) {
  w := 3.U
}
// ... and some more complex conditional
    assignments
```

Logic Can Be Expressed as a Table

- Sometimes more convenient
- Still combinational logic (gates)
- Is converted to Boolean expressions
- Let the synthesis tool do the conversion!
- We use the `switch` statement

```
switch (sel) {
  is ("b00".U) { result := "b0001".U}
  is ("b01".U) { result := "b0010".U}
  is ("b10".U) { result := "b0100".U}
  is ("b11".U) { result := "b1000".U}
}
```

The World of Combinational Logic

- With the shown operations (logic, arithmetic, Mux) all possible combinational circuits can be described
- Even the `Mux` is already *syntactic sugar*
  - A `Mux` is basically: `(a & sel) | (b & !sel)`
- But Chisel provides further constructs for more elegant description of circuits
- Stay tuned!

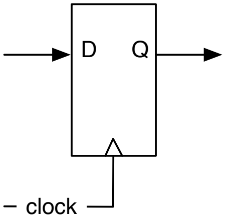
Lab 2

- Combinational circuits in Chisel
- Chisel Lab 2 Page
- Each exercise contains a test, which initially fails
- `sbt test` runs them all
  - To just run a single test, run e.g.,
    `sbt "testOnly MajorityPrinter"`

  When all test succeed you're done ;-)
- Components contain a comment where you shall add your implementation
- The initial majority example has an optional implementation in an FPGA

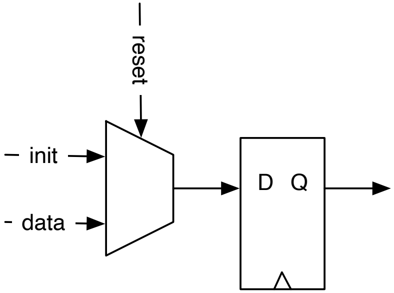
Sequential Building Blocks

- Contain a register
- Plus combinational circuits



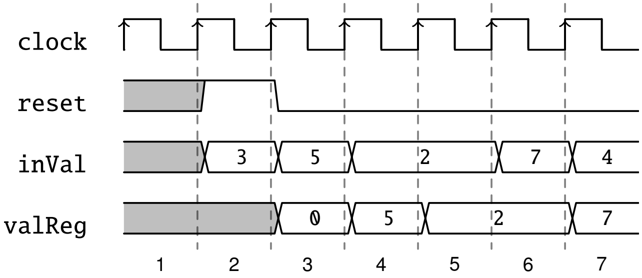
```
val q = RegNext(d)
```

Register With Reset



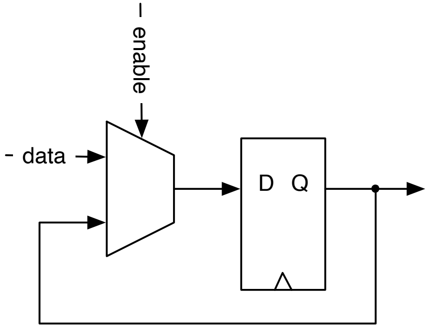
```
val valReg = RegInit(0.U(4.W))

valReg := inVal
```

Timing Diagram of the Register with Reset



- Also called waveform diagram
- Logic function over time
- Can be used to describe a circuit function
- Useful for debugging

Register with Enable



- Only when `enable` true is a value is stored

```
val enableReg = Reg(UInt(4.W))

when (enable) {
  enableReg := inVal
}
```

A Register with Reset and Enable

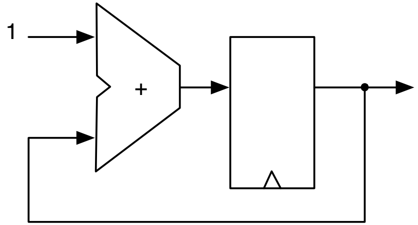
- We can combine initialization and enable

```
val resetEnableReg = RegInit(0.U(4.W))

when (enable) {
  resetEnableReg := inVal
}
```

- A register can also be part of an expression
- What does the following circuit do?

```
val risingEdge = din & !RegNext(din)
```

A Register with an Adder is a Counter



- Is a free running counter
- 0, 1, ... 14, 15, 0, 1, ...

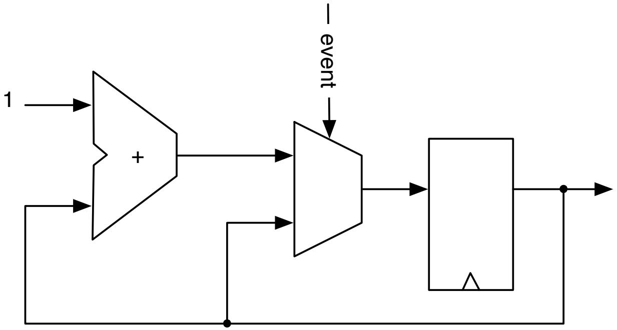
```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
```

A Counter with a Mux

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```

- This counter counts from 0 to 9
- And starts from 0 again after reaching 9
    - Starting from 0 is common in computer engineering
- A counter is the hardware version of a *for loop*
- Often needed

Counting Events



```
val cntEventsReg = RegInit(0.U(4.W))
when(event) {
  cntEventsReg := cntEventsReg + 1.U
}
```

Counting Up and Down

- Up:

```
val cntReg = RegInit(0.U(8.W))

cntReg := cntReg + 1.U
when(cntReg === N) {
  cntReg := 0.U
}
```

- Down:

```
val cntReg = RegInit(N)

cntReg := cntReg - 1.U
when(cntReg === 0.U) {
  cntReg := N
}
```

Hello World in Chisel

```
class Hello extends Module {
  val io = IO(new Bundle {
    val led = Output(UInt(1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U

  val cntReg = RegInit(0.U(32.W))
  val blkReg = RegInit(0.U(1.W))

  cntReg := cntReg + 1.U
  when(cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}
```

Reminder: We Construct Hardware

- Chisel code looks much like Java code
- But it is *not* a program in the usual sense
- It represents a circuit
- We should be able to *draw* that circuit
- The “program” constructs the circuit
- All statements are “executed” in parallel
- Statement order has mostly no meaning

Structure With Bundles

- A `Bundle` to group signals
- Can be different types
- Defined by a class that extends `Bundle`
- Named fields as `val`s within the block
- Like a C struct or VHDL record

```
class Channel() extends Bundle {
  val data = UInt(32.W)
  val valid = Bool()
}
```

Using a Bundle

- Create it with `new`
- Wrap it into a `Wire`
- Field access with *dot* notation

```
val ch = Wire(new Channel())
ch.data := 123.U
ch.valid := true.B

val b = ch.valid
```

Wire, Reg, and IO

- `UInt`, `SInt`, and `Bits` are Chisel types, not hardware
- `Wire`, `Reg`, or `IO` generates hardware
  - A `Wire` is a combinational circuit
  - A `Reg` is a register
  - A `IO` is a connection/port (for a module)
- Can wrap any Chisel type, also `Bundle` or `Vec`
- Give it a name by assigning it to a `val`

```
val number = Wire(UInt())
val reg = Reg(SInt())
```

Using = or :=

- Later assign or reassign a value or expression with `:=`

```
number := 10.U
reg := value - 3.U
```

- Note the small difference between = and `:=`
  - May be confusing to start with
- Use = when *creating* a hardware object to give it a name
- Use `:=` when assigning or reassigning to an *existing* hardware object

Vectors

- Indexable vector of elements
- Elements can be Chisel basic elements, or bundles
- Type is specified as second parameter

```
val myVec = Vec(3, SInt(10.W))
val y = myVec(2)
myVec(0) := -3.S
```

- A register file as a register of a vector

```
val vecReg = Reg(Vec(32, SInt(32.W)))
```

Components/Modules

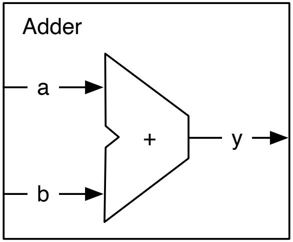
- Components/Modules are building blocks
  - Component and module are two names for the same thing
- Components have input and output ports (= pins)
  - Organized as a `Bundle`
  - Wrapped into an `IO()`
  - assigned to a field `io`
- We build circuits as a hierarchy of components
- In Chisel a component is called `Module`
- Components/Modules are used to organize the circuit
  - Similar to classes and methods in Java

Input/Output Ports

- Ports are the *interface* to a module
- Ports are bundles with directions
- Ports are used to connect modules

```
class AluIO extends Bundle {
  val function = Input(UInt(2.W))
  val inputA = Input(UInt(4.W))
  val inputB = Input(UInt(4.W))
  val result = Output(UInt(4.W))
}
```

An Adder Module



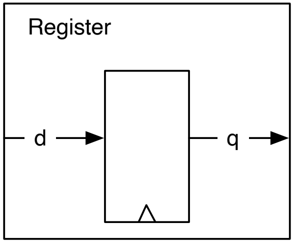
- Practically too simple, but for the slides

An Adder Module

- A `class` that `extends` `Module`
- Interface (port) is a `Bundle`, wrapped into an `IO()`, and stored in the field `io`
- Circuit description in the constructor

```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  io.y := io.a + io.b
}
```

An Register Module

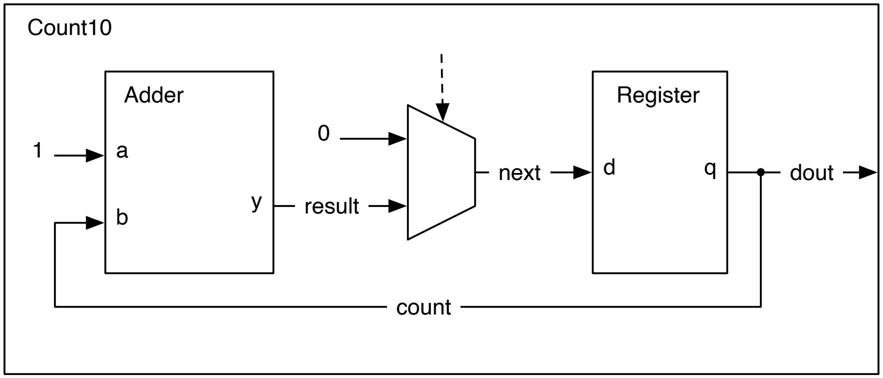


- Practically too simple, but for the slides

An Register Module

```
class Register extends Module {
  val io = IO(new Bundle {
    val d = Input(UInt(8.W))
    val q = Output(UInt(8.W))
  })

  val reg = RegInit(0.U)
  reg := io.d
  io.q := reg
}
```

An Counter out of Modules



An Counter out of Modules

```
class Count10 extends Module {
  val io = IO(new Bundle {
    val dout = Output(UInt(8.W))
  })

  val add = Module(new Adder())
  val reg = Module(new Register())

  // the register output
  val count = reg.io.q
  // connect the adder
  add.io.a := 1.U
  add.io.b := count
  val result = add.io.y
  // connect the Mux and the register input
  val next = Mux(count === 9.U, 0.U, result)
  reg.io.d := next
  io.dout := count
}
```

Using Modules/Components

- Create with `new` and wrap into a `Module()`
- Interface port via the `io` field
- Note the assignment operator `:=` on `io` fields
- Note the dot access to the field `io` and then the IO field

Chisel Main

- Create one top-level Module
- Invoke the Chisel code emitter from the App
- Pass the top module (e.g., `new Hello()`)
- Optional: pass some parameters (in an `Array`)
- Following code generates Verilog code for *Hello World*

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

Testing with Chisel

- A test contains:
  - a device under test (DUT) and
  - the testing logic
- Set input values with `poke`
- Advance the simulation with `step`
- Read the output values with `peek`
- Compare the values with `expect`
- Import following packages:

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
```

An Example DUT

- A device-under test (DUT)
- Just 2-bit AND logic

```
class DeviceUnderTest extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
    val equ = Output(Bool())
  })

  io.out := io.a & io.b
  io.equ := io.a === io.b
}
```

A ChiselTest

- Extends class `AnyFlatSpec` with `ChiselScalatestTester`
- Has the device-under test (DUT) as parameter of the `test` function
- Test function contains the test code
- Testing code can use all features of Scala

A Test

- Poke values and `expect` some output

```
class SimpleTestExpect extends AnyFlatSpec
    with ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.out.expect(0.U)
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      dut.io.out.expect(2.U)
    }
  }
}
```

Summary

- Chisel describes digital hardware embedded in Scala
- Digital design is simple: combinational circuits and registers
- We will make it more powerful with generator code
- A digital circuit is organized into components
- Components have ports with directions
- Sequential circuits are combinations of registers with combinational circuits

Lab 3

- Components and Small Sequential Circuits
- Chisel Lab 3 Page
- Each exercise contains a test, which initially fails
- `sbt test` runs them all
  - To just run a single test, run e.g.,
    `sbt "testOnly SingleTest"`
- When all tests succeed, you're done ;-)