Simple Generators

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

September 16, 2025

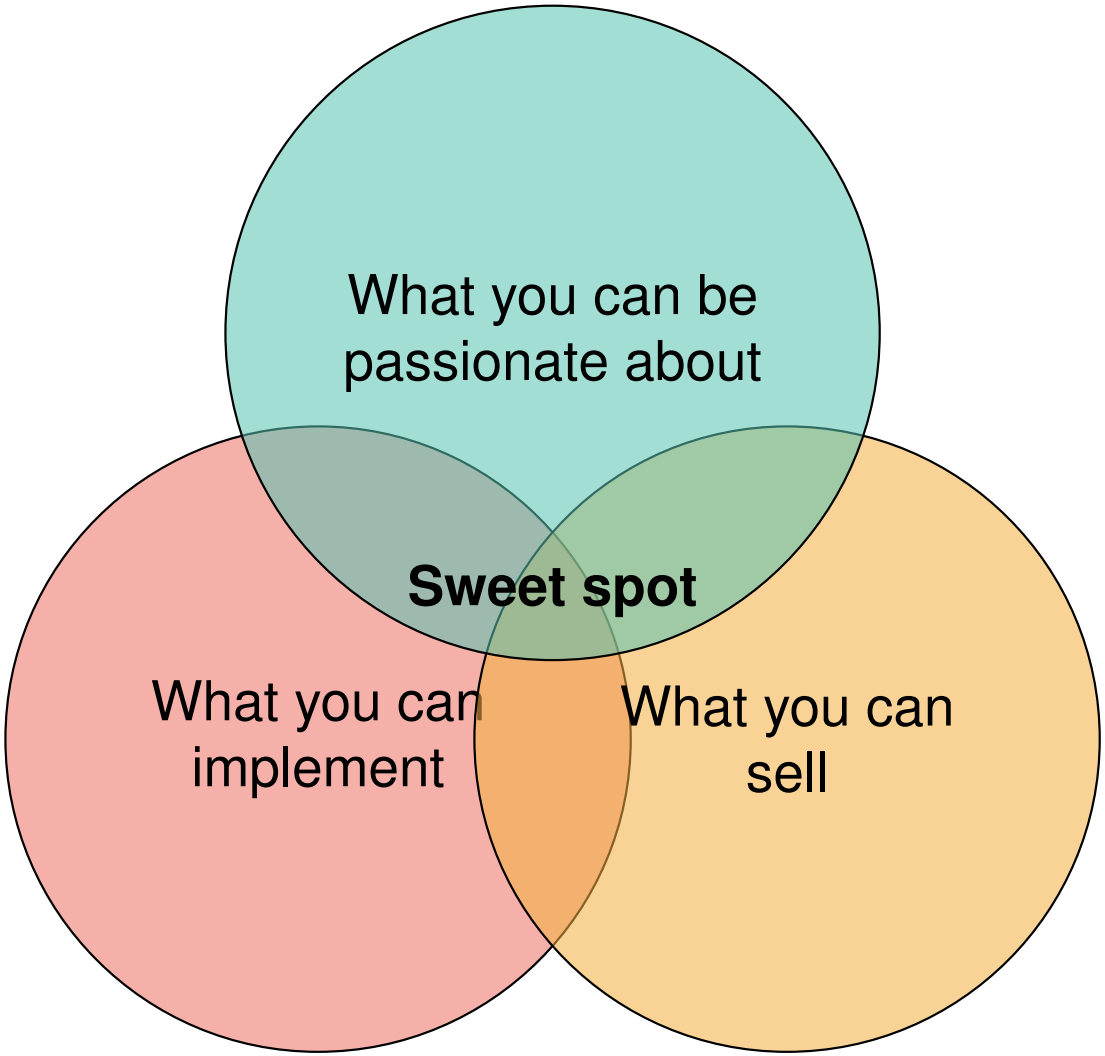
Overview

- More on agile development (Scrum)
- A bit more Scala and Chisel
- Functional programming for generators
- Hardware generators
- Object-oriented hardware design

Scrum

- Agile framework for development
- Scrum is from rugby, restarting the play
- Book by Jeff Sutherland
- Scrum The Art of Doing Twice the Work in Half the Time

Venn Diagram for Agile Development



Sprint

- One or two weeks development cycle
- Select tasks from the backlog
- Do them during the sprint
- Report after the sprint
- Even show to the customer, if possible
- Be creative about who the customer is, maybe it is me?

Things to Implement

- Collect list of things in a backlog (a TODO list)
- This is the collection for taking items for a sprint
- Prioritise things
- 80% value out of 20% feature
- The backlog list will change
- You will not do all things
- Minimum viable thing first - be able to show something

More on Agile

- Be open, share stuff. Best in the open, like GitHub (see my work)
  - I have (almost) all of my work on GitHub (slides, Chisel book, code...)
  - This could be part of your CV
- Do what you are passionate about
  - The grade should not be what you aim for
- Commit often
- Demonstrate often, let us make this a weekly thing
- Deliver value to your customer as early as possible
- Fake it until you make it

Scrum Meetings

- Have regular meetings, more often than once a week
- Scrum is a daily standup meeting
- Ask questions:
  - What did you do?
  - What have you planned to do?
  - Are there roadblocks

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

Chisel Example: 2-bit Counter

```
class Counter extends Module {
  val io = IO(new Bundle {
    val out = Output(UInt(2.W))
  })
  val count = RegInit(0.U(2.W))
  count := count + 1.U
  io.out := count
}
```

Example Test

```
test(new Counter) { c =>
  c.io.out.expect(0.U)
  c.clock.step()
  c.io.out.expect(1.U)
  c.clock.step()
  c.io.out.expect(2.U)
}
```

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
- We use Scala to program the hardware generators

Scala Summary

- Scala combines OOP and FP
- Programs are built from `object` + `main`
- `val` = immutable, `var` = mutable
- Strong static typing, but with type inference
- Functions and expressions are first-class
- Collections are central for data manipulation

Conditionals in Scala

- `if` is an expression that returns a value

```
val x = 10
val res = if (x > 0) "positive" else "non-positive"
println(res)   // "positive"
```

- `match` is similar to switch, but more powerful

```
x match {
  case 0         => "zero"
  case 1 | 2     => "one or two"
  case _         => "something else"
}
```

For Comprehensions

- A concise way to combine `map`, `filter`, `flatMap`

```
val nums = List(1, 2, 3, 4, 5)

// keep even numbers, square them
val result = for {
  x <- nums
  if x % 2 == 0
} yield x * x

println(result) // List(4, 16)
```

Pattern Matching

- Pattern matching decomposes values

```
def describe(x: Any): String = x match {
  case 0           => "zero"
  case true        => "true"
  case "hi"        => "a greeting"
  case i: Int      => "an integer: " + i
  case _           => "something else"
}

println(describe(42))   // "an integer: 42"
```

Case Classes

- Lightweight classes with built-in pattern matching

```
case class Point(x: Int, y: Int)

val p = Point(1, 2)

// pattern match on fields
p match {
  case Point(0, 0) => println("origin")
  case Point(x, y) => println(s"($x,$y)")
}
```

- Case classes are nice for collecting parameters

Pure Functions

- A pure function:
  - Always returns the same output for the same input
  - Has no side effects

```
def add(a: Int, b: Int): Int = a + b
```

- Combinational hardware modules are pure
- Side effects (state, IO, randomness) are controlled explicitly

Functions Generating Hardware

- Circuits can be encapsulated in functions
- Each *function call* generates hardware
- Simple functions can be a single line

```
def adder(v1: UInt, v2: UInt) = v1 + v2

val add1 = adder(a, b)
val add2 = adder(c, d)
```

Functional Abstraction

```
def addSub(add: Bool, a: UInt, b: UInt) =
  Mux(add, a+b, a-b)

val res = addSub(cond, a, b)

def rising(d: Bool) = d && !RegNext(d)
```

- Functions for repeated pieces of logic
- May contain state (not pure anymore)
- Functions may return *hardware*

The Counter as a Function

- Longer functions in curly brackets
- Last value is the return value

```
def counter(n: UInt) = {

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

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

val counter100 = counter(100.U)
```

Functions

- Example from Patmos execute stage

```
def alu(func: Bits, op1: UInt, op2: UInt): Bits = {
  val result = UInt(width = DATA_WIDTH)
  // some more lines...
  switch(func) {
    is(FUNC_ADD) { result := sum }
    is(FUNC_SUB) { result := op1 - op2 }
    is(FUNC_XOR) { result := (op1 ^ op2).toUInt }
    // some more lines
  }
  result
}
```

Use Functional Programming for Generators

```
def add(a: UInt, b:UInt) = a + b

val sum = vec.reduce(add)

val sum = vec.reduce(_ + _)

val sum = vec.reduceTree(_ + _)
```

- This is a simple example
- What about an arbitration tree with fair arbitration?

Functional Generation

- Anonymous functions, called *function literal*

```
(param) => function body
```

- A function for a minimum search

```
val min = vec.reduceTree((x, y) => Mux(x <
    y, x, y))
```

A Simple Tester

- Just using `println` for manual inspection

```
class SimpleTest 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()
      println("Result is: " +
          dut.io.out.peekInt())
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      println("Result is: " +
          dut.io.out.peekInt())
    }
  }
}
```

A Test with Expect

- 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)
    }
  }
}
```

Lab 4

- Write a search for the maximum circuit (with `treeReduce()`)
- Optional: add the generation of the index of the maximum value
- Emit Verilog with `emitVerilog(new Comp())`, so you can synthesize it
- Read the generated Verilog code
- Code is in lab4

Scala List for Enumeration

```
val empty :: full :: Nil = Enum(2)
```

- Can be used in wires and registers
- Symbols for a state machine

Finite State Machine

```
val empty :: full :: Nil = Enum(2)
val stateReg = RegInit(empty)
val dataReg = RegInit(0.U(size.W))

when(stateReg === empty) {
  when(io.enq.write) {
    stateReg := full
    dataReg := io.enq.din
  }
}.elsewhen(stateReg === full) {
  when(io.deq.read) {
    stateReg := empty
  }
}
```

- A simple buffer for a bubble FIFO

Component Generation

```
val cores = new Array[Module](32)

for (j <- 0 until 32)
  cores(j) = Module(new CPU())
```

- Use a Scala array, or a `Seq`, to collect components
- Generation with a Scala loop

Logic Generation

- Read a file into a table
  - E.g., to read in ROM content for a processor
- Generate a truth table algorithmically
  - E.g., generate binary to BCD translation
- Use the full power of Scala

```
val byteArray =
    Files.readAllBytes(Paths.get(fileName))
val arr = new Array[Bits](byteArray.length)
for (i <- 0 until byteArray.length) {
  arr(i) = Bits(byteArray(i), 8)
}
val rom = Vec[Bits](arr)
```

Parameterization

```
class ParamChannel(n: Int) extends Bundle {
  val data = Input(UInt(n.W))
  val ready = Output(Bool())
  val valid = Input(Bool())
}

val ch32 = new ParamChannel(32)
```

- Bundles and modules can be parametrized
- Pass a parameter in the constructor

A Module with a Parameter

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val result = Output(UInt(n.W))
  })

  val addVal = io.a + io.b
  io.result := addVal
}

val add8 = Module(new ParamAdder(8))
```

- Parameter can also be a Chisel type
- Can also be a generic type:
- `class Mod[T <: Bits](param: T) extends...`

Scala `for` Loop for Circuit Generation

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

shiftReg(0) := inVal

for (i <- 1 until 8) {
  shiftReg(i) := shiftReg(i-1)
}
```

- `for` is Scala
- This loop generates several connections
- The connections are parallel hardware

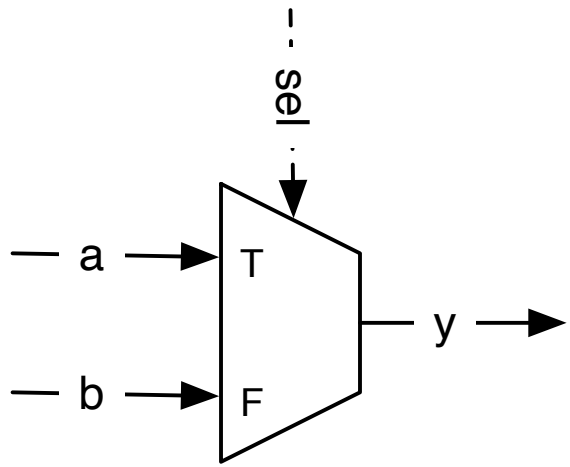
Conditional Circuit Generation

```
class Base extends Module { val io = new Bundle() }
class VariantA extends Base { }
class VariantB extends Base { }

val m = if (useA) Module(new VariantA())
        else Module(new VariantB())
```

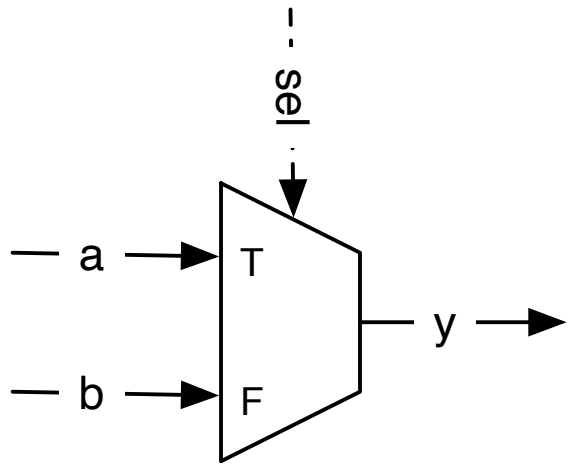
- `if` and `else` is Scala
- `if` is an expression that returns a value
  - Like "`cond ? a : b;`" in C and Java
- This is not a hardware multiplexer
- Decides which module to generate
- Could even read an XML file for the configuration

Chisel has a Multiplexer




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

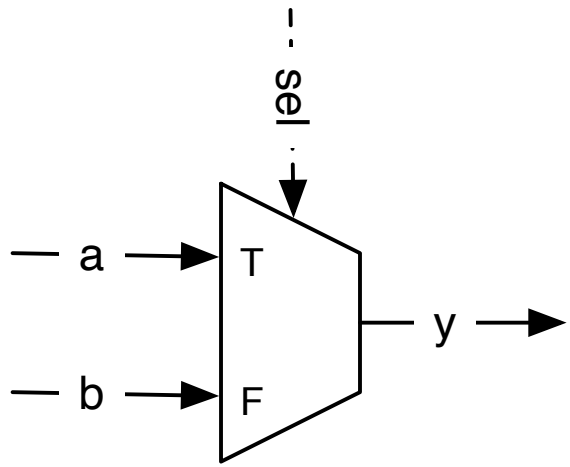
Chisel has a Multiplexer




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

- So what?
- Wait... What type is `a` and `b`?
  - Can be any Chisel type!

Chisel has a Generic Multiplexer




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

- SW people may not be impressed
- They have generics since Java 1.5 in 2004
  - `List<Flowers> != List<Cars>`

Generics in Hardware Construction

- Chisel supports generic classes with type parameters
- Write hardware generators independent of concrete type
- This is a multiplexer *generator*

```
def myMux[T <: Data](sel: Bool, tPath: T, fPath:
    T): T = {

  val ret = WireDefault(fPath)
  when (sel) {
    ret := tPath
  }
  ret
}
```

Put Generics Into Use

- Let us implement a generic FIFO
- Use the generic ready/valid interface from Chisel

```
class DecoupledIO[T <: Data](gen: T) extends
    Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen)
}
```

Define the FIFO Interface

```
class FifoIO[T <: Data](private val gen: T)
    extends Bundle {
  val enq = Flipped(new DecoupledIO(gen))
  val deq = new DecoupledIO(gen)
}
```

- We need enqueueing and dequeueing ports
- Note the `Flipped`
  - It switches the direction of ports
  - No more double definitions of an interface

But What FIFO Implementation?

- Bubble FIFO (good for low data rate)
- Double buffer FIFO (fast restart)
- FIFO with memory and pointers (for larger buffers)
  - Using flip-flops
  - Using on-chip memory
- And some more...
- This calls for object-oriented ~~programming~~ *hardware construction*

Abstract Base Class and Concrete Extension

```
abstract class Fifo[T <: Data](gen: T, val depth:
    Int) extends Module {
  val io = IO(new FifoIO(gen))

  assert(depth > 0, "Number of buffer elements
      needs to be larger than 0")
}
```

- May contain common code
- Extend by concrete classes

```
class BubbleFifo[T <: Data](gen: T, depth: Int)
    extends Fifo(gen: T, depth: Int) {
```

Select a Concrete FIFO Implementation

- Decide at hardware generation
- Can use all Scala/Java power for the decision
  - Connect to a web service, get ~~Google~~ Alphabet stock price, and decide on which to use ;-)
  - For sure a silly idea, but you see what is possible...
  - Developers may find clever use of the Scala/Java power
  - We could present a GUI to the user to select from
- We use XML files parsed at hardware generation time
- End of TCL, Python,... generated hardware

Combinational (Truth) Table Generation

```
val arr = new Array[Bits](length)
for (i <- 0 until length) {
  arr(i) = ...
}
val rom = Vec[Bits](arr)
```

- Generate a table in a Scala array
- Use that array as input for a Chisel `Vec`
- Generates a logic table at hardware construction time

Ideas for Runtime Table Generation

- Assembler in Scala/Java generates the boot ROM
- Table with a `sin` function
- Binary to BCD conversion
- Schedule table for a TDM-based network-on-chip
-
- More ideas?

Test Driven Development (TDD)

- Software development process
  - Can we learn from SW development for HW design?
- Writing the test first, then the implementation
- Started with extreme programming
  - Frequent releases
  - Accept change as part of the development
- A path to *Agile Hardware Development!*
- Not used in its pure form
  - Writing all those tests is simply considered too much work

Continuous Integration

- Run your tests on each change
- Do it also when using source control
- GitHub Actions
- I am doing it even for the Chisel book
- Code that does not contain a test *does not exist*
- Chisel community does not accept a PR without a test

Testing versus Debugging

- Debugging is during code development
- Waveform and println are easy tools for debugging
- Debugging does not help for regression tests
- Write small test cases for regression tests
- Keeps your code base *intact* when doing changes
- Better confidence in changes not introducing new bugs

Project

- You shall start organizing into groups
  - How shall we organize this?
  - Google Docs? DTU Learn? GitHub?
- Think about a project
- For inspiration, you can check Scott's students page
- It shall be a generator, not *just* hardware code
- You shall try to explore Scrum methodology
- Have those weekly meetings on reporting
- We shall have some in the class
- The report is the README.md

Lab 5

- Design a component with a generic parameter
- Emit Verilog with `emitVerilog(new Comp())` to read the generated code
- README and setup is in lab5

Summary

- Agile development is about changes – it shall be fun
- We use Scala to write hardware generators
- Next week: guest lecture on Clash for HW generators