Customized Circuit Generation

Martin Schoeberl

Technical University of Denmark

March 2, 2021

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

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

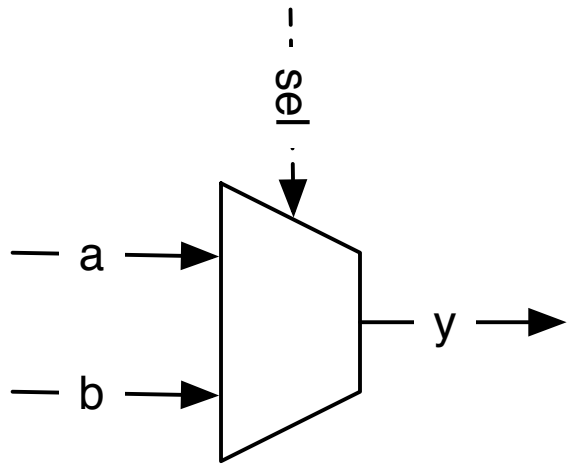
Conditional Circuit Generation

```
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

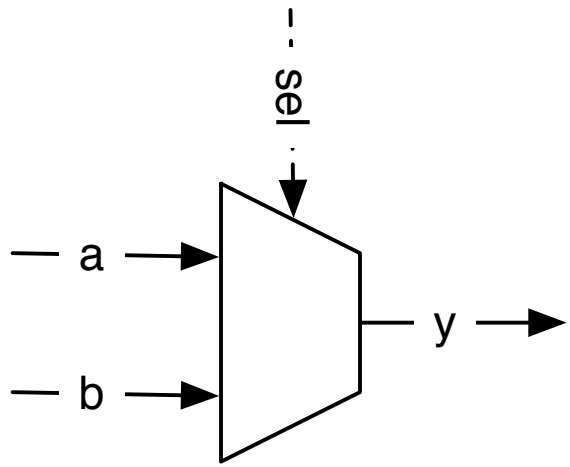
Chisel has a Multiplexer




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

- 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, 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

Binary to BCD Conversion for VHDL

```
public class GenBcdConv {

    static final int ADDRBITS = 6;
    static final int DATABITS = 8;
    static final int ROM_LEN = 1 << ADDRBITS;


    String bin(int val, int bits) {

        String s = "";
        for (int i = 0; i < bits; ++i) {
            s += (val & (1 << (bits - i - 1))) != 0 ? "1" : "0";
        }
        return s;
    }

    String getRomHeader() {

        String line = "--\n";
        line += "--\tbcdtab.vhd\n";
        line += "--\n";
        line += "--\tgenerated VHDL table for BCD conversion\n";
        line += "--\n";
        line += "--\t\tDONT edit this file!\n";
        line += "--\t\tgenerated by " + this.getClass().getName() + "\n";
        line += "--\n";
        line += "\n";
        line += "library ieee;\n";
        line += "use ieee.std_logic_1164.all;\n";
        line += "\n";
        line += "entity bcdtab is\n";
        // line +=
        // "generic (width : integer; addr_width : integer);\t-- for compatibility\n";
        line += "port (\n";
        line += "    address : in std_logic_vector(" + (ADDRBITS - 1)
            + " downto 0);\n";
        line += "    q : out std_logic_vector(" + (DATABITS - 1)
            + " downto 0)\n";
        line += ");\n";
        line += "end bcdtab;\n";
        line += "\n";
        line += "architecture rtl of bcdtab is\n";
        line += "\n";
        line += "begin\n";
        line += "\n";
        line += "process(address) begin\n";
        line += "\n";
        line += "case address is\n";

        return line;
    }

    String getRomFeet() {

        String line = "\n";
        line += "    when others => q <= \"" + bin(0, DATABITS) + "\";\n";
        line += "end case;\n";
        line += "end process;\n";
        line += "\n";
        line += "end rtl;\n";

        return line;
    }

    public void dump() {

        try {

            FileWriter romvhd = new FileWriter("bcdtab.vhd");
            romvhd.write(getRomHeader());

            for (int i = 0; i < Math.pow(2, ADDRBITS); ++i) {
                int val = ((i/10)<<4) + i%10;
                romvhd.write("    when \"" + bin(i, ADDRBITS) + "\" => q <= \""
                        + bin(val, DATABITS) + "\";");
                romvhd.write("\n");

            }

            romvhd.write(getRomFeet());
            romvhd.close();

        } catch (IOException e) {
            System.out.println(e.getMessage());
            System.exit(-1);
        }
    }

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {

        GenBcdConv la = new GenBcdConv();

        la.dump();
    }

}
```

Java Program

- Generates a VHDL table
- The core code is:

```
for (int i = 0; i < Math.pow(2, ADDRBITS); ++i) {
    int val = ((i/10)<<4) + i%10;
    // write out VHDL code for each line
```

- With all boilerplate 118 LoC

Chisel Version of Binary to BCD Conversion

```
val table = Wire(Vec(100, UInt(8.W)))
for (i <- 0 until 100) {
  table(i) := (((i/10)<<4) + i%10).U
}
val bcd = table(bin)
```

- Directly generates the hardware table as a `Vec`
- At hardware construction time
- In the same language

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 arbiter tree with fair arbitration?

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?

Memory

```
val mem = Mem(Bits(width = 8), size)

// write
when(wrEna) {
  mem(wrAddr) := wrData
}

// read
val rdAddrReg = Reg(next = rdAddr)
rdData := mem(rdAddrReg)
```

- Write is synchronous
- Read can be asynchronous or synchronous
- But there are no asynchronous memories in an FPGA

Factory Methods

- Simpler component creation and use
- Usage similar to built in components, such as `Mux`

```
val myAdder = Adder(x, y)
```

- A little bit more work on component side
- Define an `apply` method on the companion object that returns the component

```
object Adder {
  def apply(a: UInt, b: UInt) = {
    val adder = Module(new Adder)
    adder.io.a := a
    adder.io.b := b
    adder.io.result
  }
}
```

Testing and Debugging

- Nobody writes perfect code ;-)
- We need a method to improve the code
- In Java we can simply print the result:
  - `println("42");`
- What can we do in hardware?
  - Describe the whole circuit and hope it works?
  - We can switch an LED on or off
- We need some tools for debugging
- Writing testers in Chisel

Testing with Chisel

- 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 chisel3.iotesters._
```

Using peek, poke, and expect

```
// Set input values
poke(dut.io.a, 3)
poke(dut.io.b, 4)
// Execute one iteration
step(1)
// Print the result
val res = peek(dut.io.result)
println(res)

// Or compare against expected value
expect(dut.io.result, 7)
```

A Chisel Tester

- Extends class `PeekPokeTester`
- Has the device-under test (DUT) as parameter
- Testing code can use all features of Scala

```
class CounterTester(dut: Counter) extends
    PeekPokeTester(dut) {

  // Here comes the Chisel/Scala code
  // for the testing
}
```

Example DUT

- A device-under test (DUT)

```
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))
  })

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

A Simple Tester

- Just using `println` for manual inspection

```
class TesterSimple(dut: DeviceUnderTest)
    extends PeekPokeTester(dut) {

  poke(dut.io.a, 0.U)
  poke(dut.io.b, 1.U)
  step(1)
  println("Result is: " +
      peek(dut.io.out).toString)
  poke(dut.io.a, 3.U)
  poke(dut.io.b, 2.U)
  step(1)
  println("Result is: " +
      peek(dut.io.out).toString)
}
```

The Main Program for the Test

- Extend an App and invoke the `iotesters` driver
- With the DUT and the tester

```
object TesterSimple extends App {
  chisel3.iotesters.Driver(() => new
      DeviceUnderTest()) { c =>
    new TesterSimple(c)
  }
}
```

A Real Tester

- Poke values and `expect` some output

```
class Tester(dut: DeviceUnderTest) extends
    PeekPokeTester(dut) {

  poke(dut.io.a, 3.U)
  poke(dut.io.b, 1.U)
  step(1)
  expect(dut.io.out, 1)
  poke(dut.io.a, 2.U)
  poke(dut.io.b, 0.U)
  step(1)
  expect(dut.io.out, 0)
}
```

ScalaTest

- Testing framework for Scala
- `sbt` understands ScalaTest
- Run all tests: `sbt test`
- When all `expects` are ok, the test passes
- A little bit funny syntax
- Add library to `build.sbt`

```
libraryDependencies += "org.scalatest" %%
    "scalatest" % "3.0.5" % "test"
```

- Import ScalaTest library

```
import org.scalatest._
```

ScalaTest Version of our Tester

```
class SimpleSpec extends FlatSpec with Matchers {

  "Tester" should "pass" in {
    chisel3.iotesters.Driver(() => new
        DeviceUnderTest()) { c =>
      new Tester(c)
    } should be (true)
  }
}
```

Generating Waveforms

- Waveforms are timing diagrams
- Good to see many parallel signals and registers
- Additional parameters: `"--generate-vcd-output", "on"`
- IO signals and registers are dumped
- Option `--debug` puts all wires into the dump
- Generates a .vcd file
- Viewing with GTKWave or ModelSim

Call the Tester

- Using here ScalaTest
- Note `Driver.execute`
- Note `Array("--generate-vcd-output", "on")`

```
class Count6WaveSpec extends
  FlatSpec with Matchers {

  "CountWave6 " should "pass" in {
    chisel3.iotesters.Driver.
    execute(Array("--generate-vcd-output",
        "on"),() => new Count6)
    { c => new Count6Wave(c) }
    should be (true)
  }
}
```

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
- Not used in its pour form
  - Writing all those tests is simply considerer too much work

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

Summary

- Chisel is a small language
- Embedding it in Scala gives the power
- We can write circuit generators
- We can do co-simulation
- We just scratched the surface

Links to the Examples

- Example code

```
https://github.com/schoeberl/chisel-examples.git
```

- Slides

```
https://github.com/schoeberl/chisel-book.git
```

Feedback

- Plans on using Chisel?
- What went well?
- What was not so good?
- How can this Chisel course be improved?
-
- Would be happy to receive an email: `masca@dtu.dk`

Simulation Lab Session

- Testing Hello World
- Write test/verification code for a 5:1 multiplexer
- A more interesting tester on an ALU accumulator
- Project and DUT in GitHub lab1 and lab2
- https://github.com/chisel-uvm/class2020
- Extend it with ScalaTest (see Chisel book)

FPGA Lab Session

- Run and explore the UART (serial port) example
  - There is a make target that builds the UART hardware
  - `make uart`
  - It shall write some text out
  - You can observe it with a terminal (e.g., gtkterm)
- Combine the UART with a blinking LED
  - Write a ‘0’ and ‘1’ out on blink off and on
  - Best add the LEDs to `UartMain` and `uart_top.vhdl`
- Write repeated numbers 0–9 at maximum speed