Customized Circuit Generation

Martin Schoeberl

Technical University of Denmark

October 12, 2019

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

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
  }
}
```

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`

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