Chisel Tutorial read note

Yingkun Zhou
zhouyingkun15@mails.ucas.ac.cn

December 24, 2018

1.Introduce

What is chisel?

Chisel is really only a set of special class definitions, predefined objects, and usage conventions within Scala, so when you write a Chisel program you are actually writing a Scala program.

The main goal of the tutorial

The tutorial says that significant hardware designs can be completed using only the material contained herein.

The motivation of chisel

The team of chisel were motivated to develop a new hardware language by yeas of struggle e with existing hardware description languages in our research projects and hardware design courses. Verilog and VHDL were developed as hardware simulation languages, and only later did they become a basis for hardware synthesis. Much of the semantics of these languages are not appropriate for hardware synthesis and, in fact, many constructs are simply not synthesizable. Other constructs are non-intuitive in how they map to hardware implementations, or their use can accidently lead to highly inefficient hardware structures. While it is possible to use a subset of these languages and yield acceptable results, they nonetheless present a cluttered and confusing specification model, particularly in an instructional setting.

Disadvantage of Verilog and VHDL

While Verilog and VHDL include some primitive constructs for programmatic circuit generation, they lack the powerful facilities present in modern programming languages, such as object-oriented programming, type inference, support for functional programming, and reflection.

Why Scala

picked Scala not only because it includes the programming features we feel are important for building circuit generators, but because it was specifically developed as a base for domain-specific languages.

2.Hardware expressible in Chisel

The basic model

The initial version of Chisel only supports the expression of synchronous RTL (Register-Transfer Level) designs, with a single common clock. Synchronous RTL circuits can be expressed as a hierarchical composition of modules containing combinational logic and clocked state elements. Although Chisel assumes a single global clock, local clock gating logic is automatically generated for every state element in the design to save power.

Remark. *Modern hardware designs often include multiple islands of logic, where each island uses a different clock and where islands must correctly communicate across clock island boundaries. Although clock-crossing synchronization circuits are notoriously difficult to design, there are known good solutions for most scenarios, which can be packaged as library elements for use by designers. As a result, most effort in new designs is spent in developing and verifying the functionality within each synchronous island rather than on passing values between islands.*

3.Datatypes in Chisel

- SInt
- UInt
- Bool
- Bundles (similar to structs in other languages)
- Vecs (for indexable collections of values)

4.Combinational Circuits

basic model

A circuit is represented as a graph of nodes in Chisel. Each node is a hardware operator that has **zero or more** inputs and that drives **one** output.

The direct way is converting expression into a circuit tree, with named wires at the leaves and operators forming the internal nodes. But the more efficient circuit is in the shape of DAGs.

5.Builtin Operators

Bitwidth Inference Bit-width inference process will converge to a fixpoint The width of a register must be specified by the user either explicitly or from the bitwidth of the reset value or the next parameter.

We have to use triple equals === for equality and =/= for inequality to allow the native Scala equals operator to remain usable.

6.Functional Abstraction

function invoke

zero or more input and one output like the following example

```
val out = clb(a,b,c,d)
```

function definition

in order to do that, we must define the function which takes a,b,c,d as arguments and returns a wire to the output of a boolean circuit.

```
def clb(a: UInt, b: UInt, c: UInt, d: UInt): UInt =
  (a & b) | (~c & d)
```

7.Bundles and Vecs(aggregate classes)

well, it seems like that Vecs are kind of reg array.
after all, the superclass of all is data

Remark (1)**.** *Bundles and Vecs can be arbitrarily nested to build complex data structures*

Remark (2)**.** *the builtin Chisel primitive and aggregate classes do not require the new when creating an instance, whereas new user datatypes will, but not always stand.*

8.Ports

Here is the example:

```
class Decoupled extends Bundle {
  val ready = Output(Bool())
  val data = Input(UInt(32.W))
  val valid = Input(Bool())
}
```

which is a standard shackhand port type
also the port can use user defined class. In other words, folding directions into the object declarations.

```
class ScaleIO extends Bundle {
  val in = new MyFloat().asInput
  val scale = new MyFloat().asInput
  val out = new MyFloat().asOutput
}
```

the methods asInput and asOutput force all modules of the data object to the requested direction.

9.Modules

The hierarchical module namespace is accessible in downstream tools to aid in debugging and physical layout. A user-defined module is defined as a class

```
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)
}
```

Remark. *The := assignment operator, used here in the body of the definition, is a special operator in Chisel that wires the input of left-hand side to the output of the right-hand side*

module constructor function invoke

```
val m1 = Module(new Mux2())
```

10.running examples and test them

- **poke** to set input port and state values
- **step** to execute the circuit one time unit
- **peek** to read port and state values
- **expect** to compare peeked circuit values to expected arguments

Remark. *Chisel produces Firrtl intermediate representation (IR). Firrtl can be interpreted directly or can be translated into Verilog, which can then be used to generate a C++ simulator through verilator.*

To do this, on each iteration we generate appropriate inputs to the module and tell the simulation to assign these values to the inputs of the device we are testing (module or class) c, step the circuit 1 clock cycle, and test the expected value. Steps are necessary to update registers and the combinational logic driven by registers. For pure combinational paths, poke alone is sufficient to up- date all combinational paths connected to the poked input wire.

Remark. *For sequential modules we may want to delay the output definition to the appropriate time as the step function implicitly advances the clock one period in the simulation. Unlike Verilog, you do not need to explicitly specify the timing advances of the simulation; Chisel will take care of these details for you.*

what will happen in testing. By default, it will run the tests defined simulated but the Firrtl interpreter. We can instead have the module be simulated by a C++ simulator generated by Verilator by explicit typing verilator

11.State Elements

The simplest form of state element supported by Chisel is a positive edge-triggered register.

```
val reg = Reg(next = in)
```

if you think about it carefully, the state element (here I mean single variable) has two faces: one is the input and one is the output. And the output is the copy of the input signal in delayed by one clock cycle.

Remark (1)**.** *we do not have to specify the type of Reg as it will be automatically inferred from its input when instantiated in this way.*

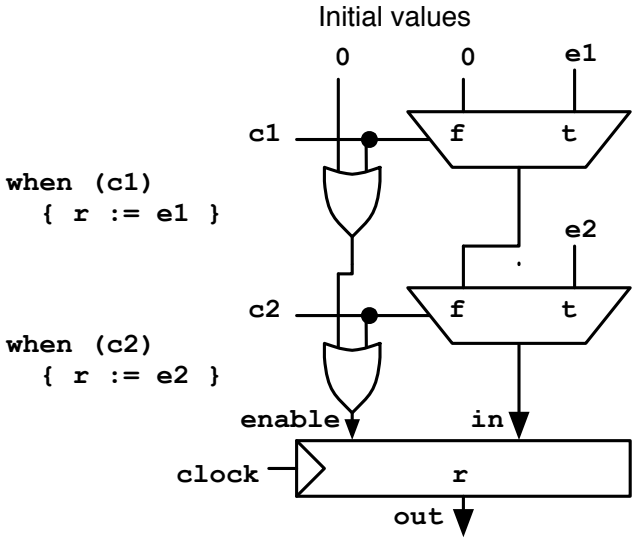
Remark (2)**.** *In the current version of Chisel, clock and reset are global signals that are implicitly included where needed*

Remark (3)**.** *the := assignment to variable say x wires an update combinational circuit on the right hand side(rhs) to the target node on the left hand side(lhs). As for reg associate circuit, the rule means that when x appears on the rhs of an assignment, its output is referenced, whereas when on lhs, its input is referenced.*

Remark (4)**.** *Purely combinational circuits cannot have cycles be- tween nodes, and Chisel will report an error if sucha cycle is detected. Just as said before, the pure combinational circuits is a DAG. Because they do not have cycles, combinational circuit can always be constructed in a feed-forward manner.* ***However****, sequential circuits naturally have feedback between nodes, and so it is sometimes necessary to reference an output wire before the producing node has defined.*

```
  val pcPlus4 = UInt()
  val brTarget = UInt()
  val pcNext = Mux(io.ctrl.pcSel, brTarget, pcPlus4)
  val pcReg = Reg(next = pcNext, init = 0.U(32.W))
  pcPlus4 := pcReg + 4.U
  ...
  brTarget := addOut
```

Remark (5)**.** *In a sequence of conditional updates, the last conditional update whose condition is true takes priority. the circuit is look like following:*



Each when statement adds anotherlevel of data mux and ORs the predicate into the enable chain. The compiler effectively adds the termination values to the end of the chain automatically

Remark. *considering update of a wire. Note that all combinational circuits need a default value.*

Memories

At most of time, we only care about SRAM. Here are two type SRAMs

```
//A one-read port, one-write port SRAM
val ram1r1w =
  Mem(1024, UInt(32.W))
```

```
val reg_raddr = Reg(UInt())
when (wen) { ram1r1w(waddr) := wdata }
when (ren) { reg_raddr := raddr }
val rdata = ram1r1w(reg_raddr)
```

```
//Single-ported SRAMs read and write conditions are mutually exclusive in the
    same when chain and write first.
val ram1p = Mem(1024, UInt(32.W))
val reg_raddr = Reg(UInt())
when (wen) { ram1p(waddr) := wdata }
.elsewhen (ren) { reg_raddr := raddr }
val rdata = ram1p(reg_raddr)
```

Remark. *Note that when constructing a memory in Chisel, the initial value of memory contents cannot be specified. Therefore, you should never assume anything about the initial contents of your Mem class.*

13.Interfaces & Bulk Connections

The ports in chisel support Subclasses and Nesting, which benefits code reuse.

```
class SimpleLink extends Bundle {
  val data = Output(UInt(16.W))
  val valid = Output(Bool())
}

class PLink extends SimpleLink {
  val parity = Output(UInt(5.W))
}

class FilterIO extends Bundle {
  val x = new PLink().flip() //???
  val y = new PLink()
}

class Filter extends Module {
  val io = IO(new FilterIO())
  ...
}
```

Bulk Connections

```
class Block extends Module {
  val io = IO(new FilterIO())
  val f1 = Module(new Filter())
  val f2 = Module(new Filter())

  f1.io.x <> io.x
  f1.io.y <> f2.io.x
  f2.io.y <> io.y
}
```

in the above way, we can compose two filters into a filter block as follows

14.Functional Module Creation

```
object Mux2 {
  def apply (sel: UInt, in0: UInt, in1: UInt) = {
    val m = new Mux2()
    m.io.in0 := in0
    m.io.in1 := in1
    m.io.sel := sel
    m.io.out
  }
}


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))
  })
  io.out := Mux2(io.sel(1),
  Mux2(io.sel(0), io.in0, io.in1),
  Mux2(io.sel(0), io.in2, io.in3))
}
```

Selecting inputs is so useful

```
Mux(c1, a, Mux(c2, b, Mux(..., default)))
MuxCase(default, Array(c1 -> a, c2 -> b, ...))
MuxCase(default,
        Array((idx === 0.U) -> a,
              (idx === 1.U) -> b, ...))
MuxLookup(idx, default,
          Array(0.U -> a, 1.U -> b, ...))
```

Remark. *the cases (eg. c1,c2) must be in parentheses.*

15.Polymorphism and Parameterization

Parameterized Functions

consider the following function:

$$y[t] = \sum_j w_j * x[t-j]$$

```
def Delays[T <: Data](x: T, n: Int): List[T] =
  if (n <= 1) List(x) else x :: Delays(RegNext(x), n-1)

def FIR[T <: Data with Num[T]](w: Seq[T], x: T): T =
  (w, Delays(x, w.length)).zipped.map(_ * _).reduce(_ + _)
```

explanation: where delays creates a list of incrementally increasing delays of its input and reduce constructs a reduction circuit given a binary combiner function f. In this case, reduce creates a summation circuit. Finally, the FIR function is constrained to work on inputs of type Num where Chisel multiplication and addition are defined.

Somehow, the circuit logic can also write in this way:

```
def Delay[T <: Bits](x: T, n: Int): T =
  if (n == 0) x else Reg(Delay(x, n - 1))

def FIR[T <: Num] (w: Array[Int], x: T) = {
  val delays = Range(0, w.length).map(i => Num(w(i)) * Delay(x, i)
  delays.foldRight(_ + _)
}
```

Parameterized Classes Like parameterized functions, we can parameterize classes to make them more reusable. For instance:

```
class FilterIO[T <: Data](type: T) extends Bundle {
  val x = type.asInput.flip // filp() ???
  val y = type.asOutput
}

class Filter[T <: Data](type: T) extends Module {
  val io = IO(new FilterIO(type))
  ...
}

val f = Module(new Filter(new PLink()))
```

another very useful example is fifo

```
class DataBundle extends Bundle {
  val A = UInt(32.W)
  val B = UInt(32.W)
}

object FifoDemo {
  def apply () = new Fifo(new DataBundle, 32)
}

class Fifo[T <: Data] (type: T, n: Int) extends Module {
  val io = IO(new Bundle {
    val enq_val = Input(Bool())
    val enq_rdy = Output(Bool())
    val deq_val = Output(Bool())
    val deq_rdy = Input(Bool())
    val enq_dat = type.asInput
```

```
    val deq_dat = type.asOutput
  })
  val enq_ptr  = Reg(init = 0.U(sizeof(n).W))
  val deq_ptr  = Reg(init = 0.U(sizeof(n).W))
  val is_full  = Reg(init = false.B)
  val do_enq   = io.enq_rdy && io.enq_val
  val do_deq   = io.deq_rdy && io.deq_val
  val is_empty = !is_full && (enq_ptr === deq_ptr)
  val deq_ptr_inc = deq_ptr + 1.U
  val enq_ptr_inc = enq_ptr + 1.U
  val is_full_next =
  Mux(do_enq && ~do_deq && (enq_ptr_inc === deq_ptr),
  true.B,
  Mux(do_deq && is_full, false.B, is_full))
  enq_ptr := Mux(do_enq, enq_ptr_inc, enq_ptr)
  deq_ptr := Mux(do_deq, deq_ptr_inc, deq_ptr)
  is_full := is_full_next
  val ram = Mem(n)
  when (do_enq) {
    ram(enq_ptr) := io.enq_dat
  }
  io.enq_rdy := !is_full
  io.deq_val := !is_empty
  ram(deq_ptr) <> io.deq_dat
}
```

It is also possible to define a generic decoupled interface:

```
class DecoupledIO[T <: Data](data: T)
   extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val data = data.cloneType.asOutput
}

class DecoupledDemo
  extends DecoupledIO()( new DataBundle ) //????

class Fifo[T <: Data] (data: T, n: Int)
   extends Module {
  val io = IO(new Bundle {
    val enq = new DecoupledIO( data ).flip()
    val deq = new DecoupledIO( data )
  })
  ...
}
```

16.further reading about chisel

- the tool that can translate the first implemented RISC-V processor trainwreck
- What's the difference between chisel 2 and chisel 3 Chisel3 vs Chisel2

- useful wiki and tutorial to introduce Chisel
- tour of scala