Chisel tutorial

Zhigang Liu

Outline

- Brief introduction
- Basic constructs
- How to learn chisel & More resources
- Today’s work

Hardware Generators

- Problem
  - hardware design is too low level
  - hard to write generators for family of designs
- Solution
  - leverage great ideas in software engineering in hardware design
  - embed hardware construction in programming language
  - leverage host language ideas and software engineering techniques
  - zero cost abstractions

Status Quo Generators

- write verilog design structurally – literal
- verilog generate command – limited
- write perl/python script that writes verilog – awkward

Chisel

- Chisel is a DSL embedded in Scala for creating and connecting circuit components, with tools for simulation and translation to Verilog.
- Best of hardware and software design ideas.
  - OOP
  - Functional
- Multiple targets(compare with llvm)
  - Chisel: Chisel/verilog -> Firrtl(Flexible Intermediate Representation for RTL)->C++/Verilog/FPGA tool flow/ASIC tool flow
  - llvm: clang/gcc -> llvm ir -> x86/mips/riscv

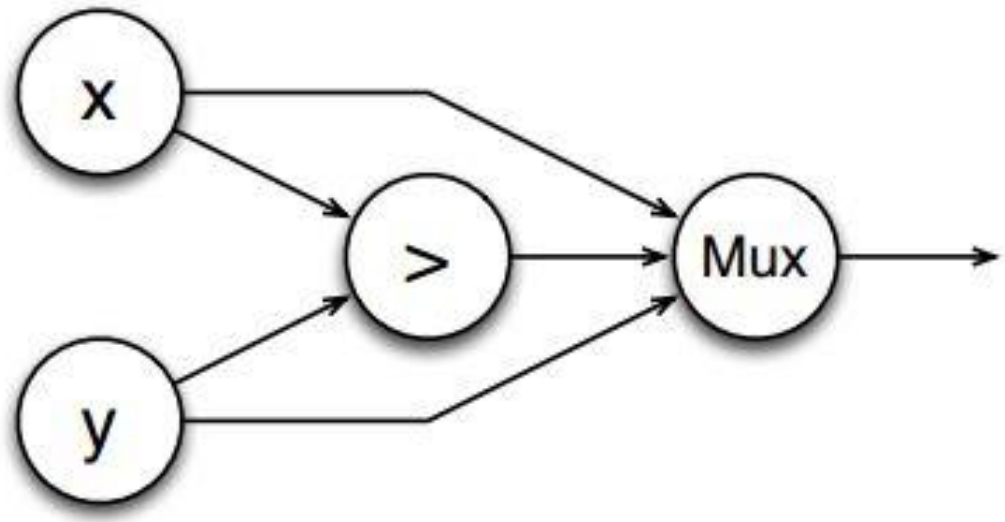
Find maximum number: Python

```
1 def max2(a, b):
2     if a > b:
3         return a
4     else:
5         return b
6
7 reduce(max2, [1, 3, 5, 7, 9])
```

Find maximum number: Chisel

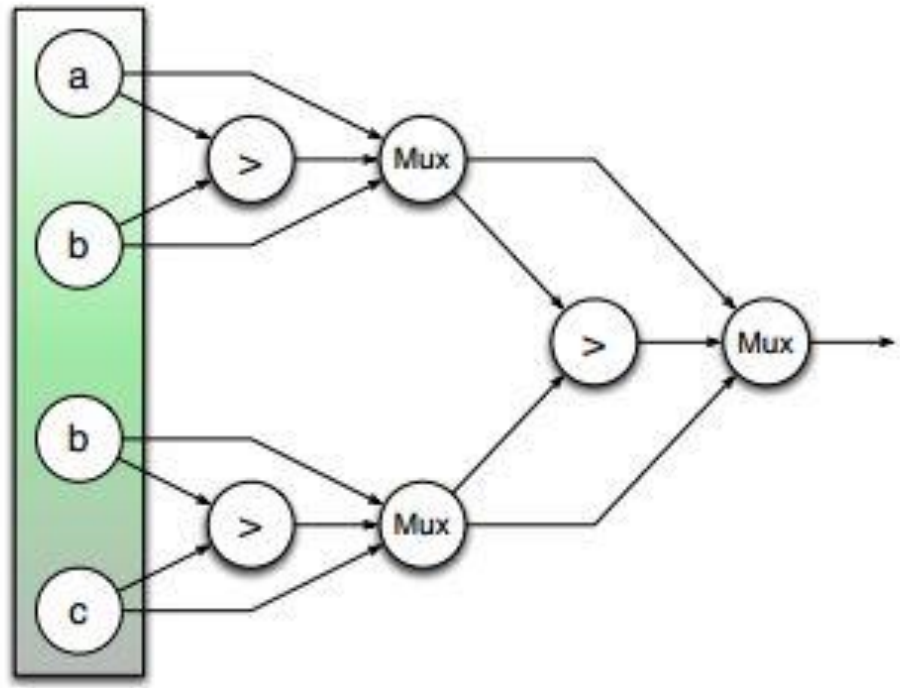
```
def Max2(x, y) = Mux(x > y, x, y)
```

```
Max2(x, y)
```



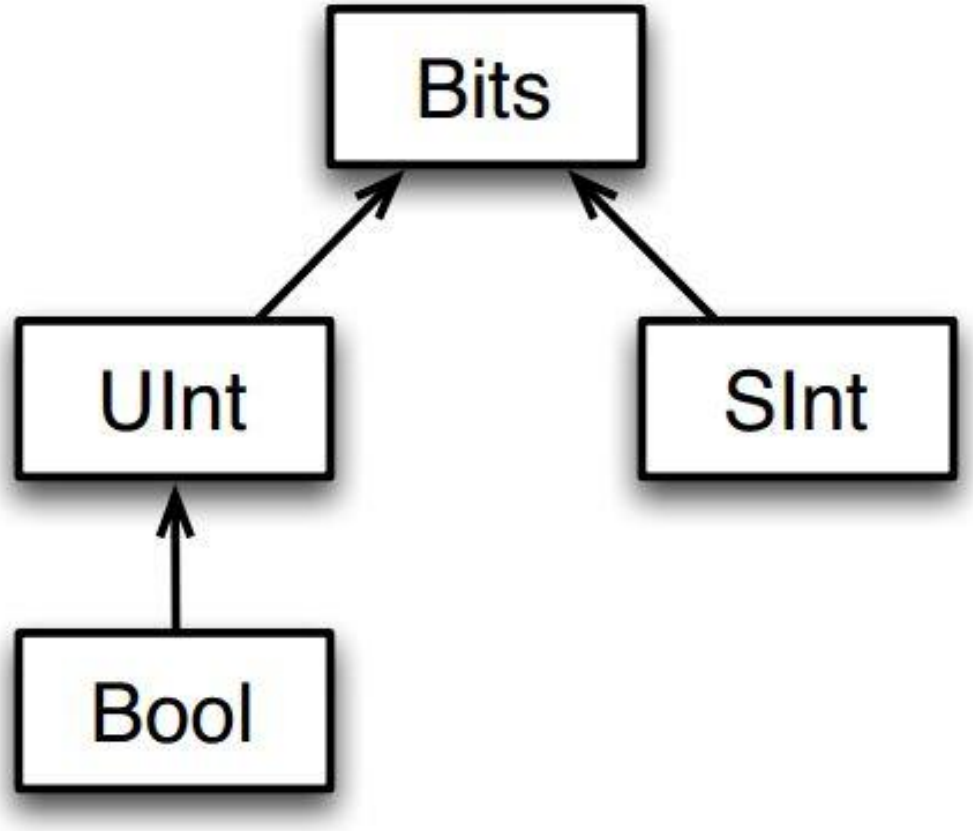
Find maximum number: Chisel

```
class MaxN(n: Int, w: Int) extends Module {
  val io = new Bundle {
    val in  = Vec.fill(n){ UInt(INPUT, w) }
    val out = UInt(OUTPUT, w)
  }
  io.out := io.in.reduceLeft(Max2)
}
```



Chisel Datatypes: basic datatypes

Basic Chisel datatypes are used to specify the type of values held in state elements or flowing on wires.



Chisel Datatypes: Bundle

Bundle : make collections of values with named fields (similar to structs in other languages

```
class MyFloat extends Bundle {
  val sign        = Bool()
  val exponent    = UInt(width = 8)
  val significand = UInt(width = 23)
}

val x  = new MyFloat()
val xs = x.sign
```

Chisel Datatypes: Vec

Vecs for indexable collections of values

constructing vecs

```
val myVec1 = Vec.fill( <number of elements> ) { <data type> }
val myVec2 = Vec(<elt0>, <elt1>, ...)
```

creating a vec of wires

```
val ufix5_vec10 = Vec.fill(10) { UInt(width = 5) }
```

creating a vec of regs

```
val reg_vec32 = Vec.fill(32){ Reg() }
```

writing

```
reg_vec32(1) := UInt(0)
```

reading

```
val reg5 = reg_vec(5)
```

Combinational Circuits

Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	`1`
`x && y`	Logical AND	`1`
`x || y`	Logical OR	`1`
`x(n)`	Extract bit, `0` is LSB	`1`
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y`
`x >> y`	Dynamic right shift	`w(x) - minVal(y`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate `x`, `n` times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If `c`, then `x`; else `y`	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	`1`
`x != y`	Inequality	`1`
`x =/= y`	Inequality	`1`
`andR(x)`	AND-reduce	`1`
`orR(x)`	OR-reduce	`1`
`xorR(x)`	XOR-reduce	`1`

`x + y`	Addition	`max(w(x),w(y))`
`x +% y`	Addition	`max(w(x),w(y))`
`x +& y`	Addition	`max(w(x),w(y))+1`
`x - y`	Subtraction	`max(w(x),w(y))`
`x -% y`	Subtraction	`max(w(x),w(y))`
`x -& y`	Subtraction	`max(w(x),w(y))+1`
`x * y`	Multiplication	`w(x)+w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y)-1)`
`x > y`	Greater than	`1`
`x >= y`	Greater than or equal	`1`
`x < y`	Less than	`1`
`x <= y`	Less than or equal	`1`
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

Modules

Defining: subclass Module with elements, code:

```
class Accum(width:Int) extends Module {
  val io = new Bundle {
    val in = UInt(INPUT,  width)
    val out  = UInt(OUTPUT, width)
  }
  val sum = new Reg(UInt())
  sum := sum + io.in
  io.out := sum
}
```

Usage: access elements using dot notation:
(code inside a Module is always running)

```
val my_module = Module(new Accum(32))
my_module.io.in := some_data
val sum := my_module.io.out
```

State Elements

Registers retain state until updated

```
val my_reg = Reg([outType:Data], [next:Data],
                 [init:Data])
```

`outType` *(optional)* register type (or inferred)
`next` *(optional)* update value every clock
`init` *(optional)* initialization value on reset

Updating: assign to latch new value on next clock:

```
my_reg := next_val
```

Read-Write Memory provide addressable memories

```
val my_mem = Mem(n:Int, out:Data)
```

`out` memory element type
`n` memory depth (elements)

Using: access elements by indexing:

```
val readVal = my_mem(addr:UInt/Int)
```

for synchronous read: assign output to `Reg`

```
mu_mem(addr:UInt/Int) := y
```

Conditional Updates

When executes blocks conditionally by `Bool`, and is equivalent to Verilog `if`

```
when(condition1) {
  // run if condition1 true and skip rest
} .elsewhen(condition2) {
  // run if condition2 true and skip rest
} .otherwise {
  // run if none of the above ran
}
```

Switch executes blocks conditionally by data

```
switch(x) {
  is(value1) {
    // run if x === value1
  } is(value2) {
    // run if x === value2
  }
}
```

Tips & More resources

- I only introduced basic language constructs here. With this, you can write ***Verilog-like*** code.
- Learn to use standard libraries, functions, helpers.
- Get to know the whole ecosystem: riscv-chisel-tutorial-bootcamp
- Handy reference: Chisel3 Cheat Sheet
- Tutorial: chisel-tutorial
- Learn by reading code: The Sodor Processor Collection
- Chisel3 Wiki
- API

Today’s work: 跑马灯

- chisel3->Verilog->bit stream
- Uses Module, Reg
- 8LEDs, light up one at a time.

Q&A