

Chisel Basic Operations

Martin Schoeberl

Technical University of Denmark

March 2, 2021

Chisel Data Types

- ▶ Data types for values on wires or state elements
- ▶ Raw collection of bits is type `Bits`
- ▶ Simple types to represent integer numbers
 - ▶ Unsigned: `UInt`
 - ▶ Signed: `SInt`
 - ▶ Subtype of `Bits`
- ▶ Automatic bit width inference
- ▶ Boolean values are of type `Bool`, a single bit value
- ▶ *Interesting* way to specify constants

`1.U`

`"habcd".U`

`"b0101".U`

`-5.S`

`true.B`

Chisel Data Types

- ▶ Bit width can be explicitly specified with a width type
 - ▶ SInt will be sign extended
 - ▶ UInt will be zero extended

UInt(8.W)

SInt(16.W)

0.U(32.W)

"habcd".U(24.W)

-5.S(16.W)

- ▶ Chisel data types are different from Scala builtin types (e.g., Scala's Int)

Bitwise Logical Operations

- ▶ Bitwise NOT, AND, OR, and XOR
- ▶ Automatic size extension to larger operand

```
val notVal = ~x
val maskOut = x & "b00001111".U
val orVal = x | y
val xorVal = x ^ y
```

- ▶ Bit reduction
- ▶ Results in a single bit

```
x.andR
x.orR
x.xorR
```

Arithmetic Operations

- ▶ Addition, subtraction, multiplication, division, modulus
- ▶ Automatic size extension to larger operand

$+$, $-$, $*$, $/$, $\%$

- ▶ Left and right shifts
- ▶ Left shift extends bit width
- ▶ Right shift reduces bit width

\ll , \gg

Bitfield Manipulations

- ▶ Extract a single bit

```
val sign = x(31)
```

- ▶ Extract a sub field from end to start position

```
val lowByte = word(7, 0)
```

- ▶ Concatenate bit fields

```
val word = Cat(highByte, lowByte)
```

Comparison

- ▶ The usual operations
 - ▶ Unusual equal and unequal operator symbols
 - ▶ To keep the original Sala operators usable for references
- ▶ Operands are UInt and SInt
- ▶ Operands can be Bool for equal and unequal
- ▶ Result is Bool

===, /=

>, >=, <, <=

Boolean Logical Operations

- ▶ Operands and result are Bool
- ▶ Logical NOT, AND, and OR

```
val notX = !x  
val bothTrue = a && b  
val orVal = x || y
```


Combinational Circuits

- ▶ Circuit is a graph of nodes
- ▶ A node is a hardware operator with zero or more inputs
- ▶ Textual expression to wire up nodes
- ▶ Named wires with some (unspecified) width

`(a | b) & ~c`

Combinational Circuits

- ▶ Simple expressions represent a circuit tree
- ▶ Arbitrary directed acyclic graphs need named subexpressions
- ▶ Using Scala's `val` keyword for variables that don't change
- ▶ Referenced multiple times

```
val cond = a & b
val result = (cond & selA) | (!cond & selB)
```

Register

- ▶ State elements
- ▶ Has it's own Chisel type `Reg`
- ▶ Positive edge triggered D flip-flop
- ▶ Synchronous reset
- ▶ Clock and reset are *hidden wires*

```
val q = RegNext(d)
```

- ▶ `d` is the input, `q` the output
- ▶ Register type is inferred by the input (`d`) type

Register

- ▶ Reset value as parameter on a `RegInit` constructor

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

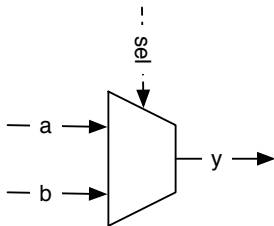
- ▶ With this forward declaration we later assign the input

```
initReg := initReg + 1.U
```

- ▶ A register can also be defined within an expression

```
val risingEdge = din & !RegNext(din)
```

A Multiplexer



- ▶ A Multiplexer selects between alternatives
- ▶ So common that Chisel provides a construct for it
- ▶ Selects `a` when `sel` is `true`. `b` otherwise
- ▶ Could be implemented with conditional updates
- ▶ Automagical type selection on input types

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

A Small Circuit

- ▶ Our Chisel knowledge is complete enough to implement any digital circuit
- ▶ Maybe not in the most elegant way ;-)
- ▶ A counter is a simple basic component
- ▶ The following counts from 0 to 100

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

```
cntReg := Mux(cntReg === 100.U,  
              0.U, cntReg + 1.U)
```

The Complete Counter Module

```
class Counter extends Module {  
  val io = IO(new Bundle {  
    val cnt = Output(UInt(8.W))  
  })  
  
  val cntReg = RegInit(0.U(8.W))  
  
  cntReg := Mux(cntReg === 100.U,  
    0.U, cntReg + 1.U)  
  
  io.cnt := cntReg  
}
```

Data Aggregation

- ▶ A Bundle groups several named fields
- ▶ Like a C struct or VHDL record
- ▶ Vec is a vector of elements with the same type
- ▶ Can be arbitrary mixed

```
class AluFields extends Bundle {  
  val function = UInt(2.W)  
  val inputA = UInt(8.W)  
  val inputB = UInt(8.W)  
  val result = UInt(8.W)  
}
```


Vectors

- ▶ Indexable vector of elements
- ▶ Elements can be Chisel basic elements, or bundles
- ▶ Type is specified as second parameter

```
val myVec = Vec(3, SInt(10.W))  
val y = myVec(2)  
myVec(0) := -3.S
```

- ▶ A register file as a register of a vector

```
val vecReg = Reg(Vec(32, SInt(32.W)))
```

Ports

- ▶ Ports used to connect modules
- ▶ Ports are bundles with directions

```
class AluIO extends Bundle {  
  val function = Input(UInt(2.W))  
  val inputA = Input(UInt(4.W))  
  val inputB = Input(UInt(4.W))  
  val result = Output(UInt(4.W))  
}
```

Port Directions

- ▶ Can be assigned at instantiation

```
class ExecuteIO extends Bundle {  
  val dec = Input(new DecodeExecute())  
  val mem = Output(new ExecuteMemory())  
}
```

Port Directions

- ▶ Can be reversed with the Flipped
- ▶ Convenient to have one bundle definition working as source and destination used between two modules

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

```
class ChannelUsage extends Bundle {  
  val input = new Channel()  
  val output = Flipped(new Channel())  
}
```

Modules

- ▶ Modules are used to organize the circuit
- ▶ Similar to VHDL components (entity/architecture)
- ▶ A class that inherits from `Module`
- ▶ Circuit description in the constructor
- ▶ Interface (port) is a `Bundle`, wrapped into an `IO()`, and stored in the field `io`

```
class Adder extends Module {  
    val io = IO(new Bundle {  
        val a = Input(UInt(4.W))  
        val b = Input(UInt(4.W))  
        val result = Output(UInt(4.W))  
    })  
  
    val addVal = io.a + io.b  
    io.result := addVal  
}
```

Module Usage

- ▶ Create with `new` and wrap into a `Module()`
- ▶ Interface port via the `io` field
- ▶ Note the assignment operator `:=` on `io` fields

```
val adder = Module(new Adder())  
adder.io.a := ina  
adder.io.b := inb  
val result = adder.io.result
```

Conditional Assignments

- ▶ Conditional update of a value
- ▶ Combinational circuit needs to be declared as a `Wire`
- ▶ Last assignment counts
- ▶ Is basically a multiplexer

```
val v = Wire(UInt())  
v := 5.U  
when (condition) {  
    v := 0.U  
}
```

```
when (c1) { v := 1.U }  
when (c2) { v := 2.U }
```

The Counter With a Conditional Update

```
class Counter2 extends Module {  
  val io = IO(new Bundle {  
    val cnt = Output(UInt(8.W))  
  })  
  
  val cntReg = RegInit(0.U(8.W))  
  
  cntReg := cntReg + 1.U  
  when (cntReg === 100.U) {  
    cntReg := 0.U  
  }  
  
  io.cnt := cntReg  
}
```


Chained Conditionals

- ▶ Chain of conditionals with `.elsewhen`
- ▶ With an optional *else* path with `.otherwise`
- ▶ Note that Scala has `if/else`
 - ▶ Does NOT result in hardware
 - ▶ Are used to conditionally *generate* hardware
 - ▶ We will look at this later
- ▶ Note the “.” at the operators

```
when (c1) { v := 1.U }  
.elsewhen (c2) { v := 2.U }  
.otherwise { v := 3.U }
```

Switch Statement

- ▶ Series of comparisons
- ▶ Chisel allows combinational logic be updated conditionally
- ▶ Chisel disallows incomplete specified logic (= latches)
- ▶ Chisel will report a runtime error

```
switch(fn) {  
  is(0.U) { result := a + b }  
  is(1.U) { result := a - b }  
  is(2.U) { result := a | b }  
  is(3.U) { result := a & b }  
}
```

More Chisel Example Code

- ▶ The time-predictable processor Patmos
- ▶ An SRAM controller for the DE2-115 board
- ▶ An SSRAM controller
- ▶ An UART
- ▶ A memory arbiter
- ▶ Caches
- ▶ ...
- ▶ <https://github.com/t-crest/patmos>

More Chisel Documentation

- ▶ Textbook “Digital Design with Chisel”
- ▶ 2nd edition is out
- ▶ <https://github.com/schoeberl/chisel-book>
- ▶ Feedback is welcome
- ▶ Contains all the slides

Very Minimal Hello World

```
class Hello extends Module {  
  val io = IO(new Bundle {  
    val out = Output(UInt(8.W))  
  })  
  io.out := 42.U  
}
```

- Produces hardware for a single constant

Testing the Minimal Hello World

```
class HelloTests(c: Hello) extends  
  PeekPokeTester(c) {  
    step(1)  
    expect(c.io.out, 42)  
  }
```

- ▶ Drive the simulation with `step(1)`, which is a single clock tick
- ▶ Test output against expected value

Link to Some Examples

- ▶ Example code
- ▶ <https://github.com/schoeberl/chisel-examples.git>

Simulation Lab Session

- ▶ Combinational circuits
- ▶ <https://github.com/schoeberl/chisel-lab/tree/master/lab2>

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