Chisel3 Cheat Sheet

Version 0.5 (beta): August 4, 2019

Notation In This Document:
For Functions and Constructors:
Arguments given as `kwd:type` (name and type(s))
Arguments in brackets (`[...]`) are optional.
For Operators:
`c`, `x`, `y` are Chisel `Data`; `n`, `m` are Scala `Int`
`w(x)`, `w(y)` are the widths of `x`, `y` (respectively)
`minVal(x)`, `maxVal(x)` are the minimum or maximum possible values of `x`

Basic Chisel Constructs

Chisel Wire Operators:

```
// Allocate a as wire of type UInt()
val x = Wire(UInt())
x := y // Connect wire y to wire x
```

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
  }
}
```

Enum generates value literals for enumerations

`val s1::s2:: ... ::sn::Nil`
`= Enum(nodeType:UInt, n:Int)`

`s1, s2, ..., sn`	will be created as `nodeType` literals with distinct values
`nodeType`	type of `s1, s2, ..., sn`
`n`	element count

Math Helpers:

`log2Ceil(in:Int): Int`	log_2(`in`) rounded up
`log2Floor(in:Int): Int`	log_2(`in`) rounded down
`isPow2(in:Int): Boolean`	`True` if `in` is a power of 2

Basic Data Types

Constructors:

`Bool()`	type, boolean value
`true.B` or `false.B`	literal values
`UInt(32.W)`	type 32-bit unsigned
`UInt()`	type, width inferred
`77.U` or `"hdead".U`	unsigned literals
`1.U(16.W)`	literal with forced width
`SInt()` or `SInt(64.W)`	like UInt
`-3.S` or `"h-44".S`	signed literals
`3.S(2.W)`	signed 2-bits wide *value -1*

Bits, UInt, SInt Casts: reinterpret cast except for:

`UInt --> SInt`	Zero-extend to SInt

State Elements

Registers retain state until updated

`val my_reg = Reg(UInt(32.W))`

Flavors

`RegInit(7.U(32.w))`	reg with initial value 7
`RegNext(next_val)`	update each clock, no init
`RegEnable(next,enable)`	update, with enable gate

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`

Modules

Defining: subclass `Module` with elements, code:

```
class Accum(width:Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out = Output(UInt(width.W))
  })
  val sum = 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
```

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`
`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`

UInt bit-reduction methods:

Chisel	Explanation	Width
`x.andR`	AND-reduce	1
`x.orR`	OR-reduce	1
`x.xorR`	XOR-reduce	1

As an example to apply the andR method to an SInt use x.asUInt.andR

Hardware Generation

Functions provide block abstractions for code. Scala functions that instantiate or return Chisel types are code generators.

Also: Scala's `if` and `for` can be used to control hardware generation and are equivalent to Verilog `generate if/for`

```
val number=Reg(if(can_be_negative) SInt()
else UInt())
```

will create a Register of type SInt or UInt depending on the value of a
Scala variable

Aggregate Types

Bundle contains `Data` types indexed by name

Defining: subclass `Bundle`, define components:

```
class MyBundle extends Bundle {
  val a = Bool()
  val b = UInt(32.W)
}
```

Constructor: instantiate `Bundle` subclass:

`val my_bundle = new MyBundle()`

Inline defining: define a `Bundle` type:

```
val my_bundle = new Bundle {
  val a = Bool()
  val b = UInt(32.W)
}
```

Using: access elements through dot notation:

```
val bundleVal = my_bundle.a
my_bundle.a := Bool(true)
```

Vec is an indexable vector of `Data` types

`val myVec = Vec(elts:Iterable[Data])`

`elts` initial element `Data` (vector depth inferred)

`val myVec = Vec.fill(n:Int) gen:Data`

`n` vector depth (elements)

`gen` initial element `Data`, called once per element

Using: access elements by dynamic or static indexing:

`readVal := myVec(ind:Data/idx:Int)`

`myVec(ind:Data/idx:Int) := writeVal`

Functions: (`T` is the `Vec` element's type)

`.forall(p:T=>Bool): Bool`	AND-reduce `p` on all elts
`.exists(p:T=>Bool): Bool`	OR-reduce `p` on all elts
`.contains(x:T): Bool`	`True` if this contains `x`
`.count(p:T=>Bool): UInt`	count elts where `p` is `True`
`.indexWhere(p:T=>Bool): UInt`	
`.lastIndexWhere(p:T=>Bool): UInt`	
`.onlyIndexWhere(p:T=>Bool): UInt`	

Standard Library: Function Blocks

Stateless:

`PopCount(in:Bits/Seq[Bool]): UInt`
Returns number of hot (= 1) bits in `in`

`Reverse(in:UInt): UInt`
Reverses the bit order of `in`

`UIntToOH(in:UInt, [width:Int]): Bits`
Returns the one-hot encoding of `in`
`width` *(optional, else inferred)* output width

`OHToUInt(in:Bits/Seq[Bool]): UInt`
Returns the `UInt` representation of one-hot `in`

`Counter(n:Int]): UInt`
.inc() bumps counter returning true when n reached
.value returns current value

`PriorityEncoder(in:Bits/Iterable[Bool]): UInt`
Returns the position the least significant `1` in `in`

`PriorityEncoderOH(in:Bits): UInt`
Returns the position of the hot bit in `in`

`Mux1H(in:Iterable[(Data, Bool]): Data`
`Mux1H(sel:Bits/Iterable[Bool], in:Iterable[Data]): Data`
`PriorityMux(in:Iterable[(Bool, Bits]): Bits`
`PriorityMux(sel:Bits/Iterable[Bool], in:Iterable[Bits]): Bits`
A mux tree with either a one-hot select or multiple selects (where the first inputs are prioritized)
`in` iterable of combined input and select`(Bool, Bits)` tuples or just mux input `Bits`
`sel` select signals or bitvector, one per input

Stateful:

`LFSR16([increment:Bool]): UInt`
16-bit LFSR (to generate pseudorandom numbers)
`increment` *(optional, default True)* shift on next clock

`ShiftRegister(in:Data, n:Int, [en:Bool]): Data`
Shift register, returns `n`-cycle delayed input `in`
`en` *(optional, default True)* enable

Standard Library: Interfaces

DecoupledIO is a `Bundle` with a ready-valid interface

Constructor:

`Decoupled(gen:Data)`
`gen` Chisel `Data` to wrap ready-valid protocol around

Interface:
(in) `.ready` ready `Bool`
(out) `.valid` valid `Bool`
(out) `.bits` data

ValidIO is a `Bundle` with a valid interface

Constructor:

`Valid(gen:Data)`
`gen` Chisel `Data` to wrap valid protocol around

Interface:
(out) `.valid` valid `Bool`
(out) `.bits` data

Queue is a `Module` providing a hardware queue

Constructor:

`Queue(enq:DecoupledIO, entries:Int)`
`enq` `DecoupledIO` source for the queue
`entries` size of queue

Interface:
`.io.enq` `DecoupledIO` source (flipped)
`.io.deq` `DecoupledIO` sink
`.io.count` `UInt` count of elements in the queue

Pipe is a `Module` delaying input data

Constructor:

`Pipe(enqValid:Bool, enqBits:Data, [latency:Int])`
`Pipe(enq:ValidIO, [latency:Int])`
`enqValid` input data, valid component
`enqBits` input data, data component
`enq` input data as `ValidIO`
`latency` *(optional, default 1)* cycles to delay data by

Interface:
`.io.enq` `ValidIO` source (flipped)
`.io.deq` `ValidIO` sink

Arbiters are `Module`s connecting multiple producers to one consumer

`Arbiter` prioritizes lower producers
`RRArbiter` runs in round-robin order

Constructor:

`Arbiter(gen:Data, n:Int)`
`gen` data type
`n` number of producers

Interface:
`.io.in` `Vec` of `DecoupledIO` inputs (flipped)
`.io.out` `DecoupledIO` output
`.io.chosen` `UInt` input index on `.io.out`, does not imply output is valid