Chisel 3.6 Cheatsheet (v0) (page 1)

For Operators: `c`, `p`, `x`, `y` are Chisel Data; `n`, `m` are Scala Ints `w(x)`, `w(y)` are the widths of x, y (respectively)
`minVal(x)`, `maxVal(x)` are the minimum or maximum possible values of x

Chisel Keywords at a glance

Bool, UInt, SInt, ChiselEnum, Clock, Reset, AsyncReset, Bundle, Vec, Record
Reg, Mem, Wire, IO — **Input, Output, Flipped**
when, elsewhen, otherwise, switch, is — **Module, RawModule, ExtModule**

Use Scala val to create wires, instances, etc.

```
val x = Wire(UInt())
val y = x                  // drive wire y from wire x
```

More constructors	Explanation
`Bool()`	Bool generator
`true.B or false.B`	Bool literals
`UInt()`	Unsigned integer
`UInt(32.W)`	Unsigned integer 32 bit
`29.U(6.W)`	Unsigned literal 29 6 bits
`"hdead".U`	Unsigned literal 0xDEAD 16 bits
`BigInt("12346789ABCDEF", 16).U` (Don't use Scala Int to create large literals)	Make large UInt literal
`SInt()`	Signed integer
`SInt(64.W)`	Signed integer 64 bit
`-3.S`	Signed integer literal
`3.S(2.W)`	Signed 2-bit value (-1)

Construct State elements, registers

Constructor	Explanation
`Reg(UInt())`	Creates a UInt register
`RegInit(7.U(32.W))`	32-bit Reg with initial value of 7
`RegNext(nextValue)`	Reg updated on rising clock, no initial value
`RegNext(nextValue, 3.U(32.W))`	Reg updated on rising clock but with initial value 3
`RegEnable(nextVal, enable)`	Reg updated on rising clock with an enable gate
`RegEnable(next, init, enable)`	Reg updated on rising clock with an init and enable

ChiselEnum

```
object Op extends ChiselEnum {
  val load = Value(0x03.U)
  val imm = Value(0x13.U)
  . . .
  val jal = Value(0x6f.U)
}
```

```
when (foo === Op.load)
{...}
.elsewhen(foo===Op.imm)
{...}
```

Aggregates – Bundles and Vecs

Anonymous bundle

```
val myB = new Bundle(
  val myBool = Bool()
  val myInt = UInt(5.W)
)
```

Bundle class

```
class MyBundle extends Bundle {
  val myBool = Bool()
  val myInt = UInt(5.W)
}
val myB = new MyBundle
```

Extending a Bundle

```
class MyExtendedBundle
  extends MyBundle {
  val newField = UInt(10.W)
}
```

Bundle with directions (default direction is Output)

```
Class MyBundle extends Bundle {
  val in = Input(Bool())
  val myInt = Output(UInt(5.W))
}
```

Create IO from bundle

```
val x = IO(new MyBundle)
```

Recursively flip input/output in io

```
Val fx = IO(Flipped(new MyBundle)
```

Coerce direction to all the same direction

```
val fx = IO(Output(new MyBundle)
```

Access elements via dots

```
val int1 = myB.myInt
myB.myBool := true.B
```

Vec Constructors:

Vec(size: Int, typeGen: **Data**) // create a vec of typeGen
VecInit.fill(size: Int, hwGen: Data) // create Vec and initialize to hwGen
VecInit.tabulate(size: Int) { i => hwGen: Data } // hwGen can use i for element creation
Note: Always create Reg(Vec*(...)), never Vec*(...Reg()) (reg of vec not vec of reg)

Accessing Vec elements

```
x := myVec(index: UInt) or myVec(index: Int)
myVec(index: UInt) or myVec(index: Int) := y
```

Example: vec init of a register file of 31 UInt registers of 32bits width

```
val regfile = RegInit(VecInit(Seq.fill(31)(0.U(32.W))))
```

Special methods on Vec	Explanation
`.forall(p: T => Bool): Bool`	AND-reduce p on all elts
`.exists(p: T => Bool): Bool`	Bool literals
`.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`	index where p is true.B
`.lastIndexWhere (p: T => Bool): UInt`	last index where p is true.B
`.onlyIndexWhere(p: T => Bool): UInt`	last index where p is true.B

Connections	Explanation (c is consumer, p is producer)
`c := p`	Basic connect, p drives c
`c :#= p`	(coercing mono-direction): connects all members of p to c; regardless of alignment
`c :<= p`	(aligned-direction): connects all aligned (non-flipped) c members from p
`c :>= p`	(flipped-direction): connects all flipped p members from c
`c :<>= p`	(bi-direction operator): connects all aligned c members from p; all flipped p members from c
`.squeeze`	allow truncation
`.waive`	allow missing connections

Operators on data.

Operator	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(hi, lo)`	Extract bitfield	hi - lo +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`	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
`x.andR`	AND-reduce	1
`x.orR`	OR-reduce	1
`x.xorR`	XOR-reduce	max(w(x), w(y))

Chisel 3.6 Cheatsheet (v0) (page 2)

Chisel Code 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's if/for

val number = Reg(if(canBeNegative) SInt() else UInt ())
will create a Register of type SInt or UInt depending on the value of a Scala variable.

Use the 'when' construct instead of individual muxing

```
when(condition1) {
  x := y
}.elseWhen (condition2) {
  x := x & y
  z := y
}.otherwise {
  x := z
}
```

Use when construct instead of individual muxing

```
class Delay(n: Int, payloadType: Data) extends Module {
  val in = IO(Input(payloadType))
  val out = IO(Output(payloadType))
  out := (0 until n).foldLeft(in) {
    case (last, x) => RegNext(last) }
}
```

Parameterize Modules

Insert n registers between input and output of type payload, using scala collection methods

```
class Delay(n: Int, payloadType: Data) extends Module {
  val in = IO(Input(payloadType))
  val out = IO(Output(payloadType))
  out := (0 until n).foldLeft(in) {
    case (last, x) => RegNext(last) }
}
```

Instantiate and connect to SubModules

```
class Parent(n: Int, payloadType: Data) extends Module {
  val in = IO(Input(payloadType))
  val out = IO(Output(payloadType))
  val child = Module(new Delay(n, payloadType))
  child.in := in
  out := child.out
}
```

Casting

`.asTypeOf` is hardware cast (works for HW data or Chisel Types)

```
0.U.asTypeOf(new MyBundle())
```

`chiselTypeOf(...)` copy type of HW along with parameters and directions

```
val foo = IO(chiselTypeOf(bar))
```

Standard Library – Bit checks.

Operator	Return	Explanation
`PopCount(in:Bits)`	UInt	number of hot (= 1) bits in `in`
`PopCount(in:Seq[Bool])`	UInt	number of hot (= 1) bits in `in`
`Reverse(in:UInt)`	UInt	Reverses the bit order of `in`
`UIntToOH(in:UInt, [width:Int])`	Bits	the one-hot encoding of `in` width (optional, else inferred) output width
`OHToUInt(in:Bits)`	UInt	the UInt representation of one-hot `in`
`OHToUInt(in: Seq[Bool])`	UInt	the UInt representation of one-hot `in`
`PriorityEncoder(in:Bits)`	UInt	the position the least significant 1 in `in`
`PriorityEncoder(in:Iterable[Bool])`	UInt	the position the least significant 1 in `in`
`PriorityEncoderOH(in:Bits)`	UInt	the position of the hot bit in `in`
`Mux1H(sel: Seq[Bool], in: Seq[Data])`	Data	One hot mux
`Mux1H(sel: UInt, Seq[Data])`	Data	One hot mux
`Mux1H(sel:UInt, in: UInt)`	Data	One hot mux
`PriorityMux(in:Iterable[(Bool,Bits)])`	Bits	Priority Mux
`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)

Standard Library – Stateful.

Counter(n:Int): Counter (simple)

Example:

```
val c = new Counter(n)
val wrap = WireInit(false.B)
when(cond) { wrap := c.inc() } // .inc returns true when wrap occurs
```

Counter(cond: UInt, n:Int): (UInt, Bool)

Example:

```
val countOn = true.B // increment counter every clock cycle
val (counterValue, counterWrap) = Counter(countOn, 4)
when (counterValue === 3.U) {
  ...
}
```

Counter(r: Range, enable: Bool, reset: Bool) (UInt, Bool)

Example:

```
val (counterValue, counterWrap) = Counter(0 until 10 by 2)
when (counterValue === 4.U) {
  ...
}
```

LFSR(width: Int, increment: Bool, seed: Option[BigInt]): UInt

Example:

```
val pseudoRandomNumber = LFSR(16)
```

ShiftRegister(in: Data, n: Int[, en: Bool]): Data add **n** registers

Example:

```
val regDelayTwo = ShiftRegister(nextVal, 2, ena)
```

Standard Library – Interfaces.

DecoupledIO(gen: Data): Wrap bundle with a ready valid interface

Interface: .ready read only Bool,.valid Bool, .bits payload data

ValidIO(gen: Data) Wrap **gen** with a valid interface

Interface: .valid Bool, .bits

Queue(gen: Data, entries: Int, pipe: Boolean, flow: Boolean, useSyncReadMem: Boolean, hasFlush: Boolean)

Interface: io.enq: Flipped(ReadyValid[gen]), io.deq: ReadyValid[gen]

Example:

```
val q = (new Queue(UInt(), 16))
q.io.enq <> producer.io.out
consumer. Module io.in <> q.io .deq
```

Pipe(enqValid:Bool, enqBits:Data, [latency:Int]) or **Pipe(enq:ValidIO, [latency:Int]):** Module delaying input

Interface: io.in: Flipped(ReadyValid[gen]), io.out: ReadyValid[gen]

Example:

```
val foo = Module(new Pipe(UInt(8.W)), 4)
pipe.io.enq := producer.io
consumer.io := pipe.io.deq
```

Arbiter(gen: Data, n: Int): Connect multiple producers to one consumer

Interface: io.in Vec of inputs (Flipped(ReadyValid[gen]), io.out: ReadyValid[gen])

Variants: RRArbiter (round robin) **LockingArbiter**

Example:

```
val arb = Module(new Arbiter(UInt(), 2))
arb.io.in(0) <> producer0.io.out
arb.io.in(1) <> producer1.io.out
consumer.io.in <> arb.io.out
```

Definition & Instance

```
@instantiable
class Child extends Module {
  @public val in = IO(Input(Bool()))
  @public val out = IO(Output(Bool()))
  out := in
}

class Parent(n: Int, payloadType: Data) extends Module {
  val childDef = Definition(new Delay(n, payloadType))
  val in = IO(Input(payloadType))
  val out = IO(Output(payloadType))
  val child = Instance(childDef)
  child.in := in
  out := child.out
}
```