Chisel

Chapter 2

Data types

```
Bits(8.W)
UInt(8.W)
SInt(10.W)
Bool()
```

Width

```
n.W
Bits(n.W)
```

Constants

```
0.U
-3.S
3.U(4.W)
"hff".U
"o377".U
"b1111_1111".U
'A'.U
true.B
false.B
```

Combinatorial circuits

for `UInt`, `SInt` and `Bool`:

```
val and = a & b
val or = a | b
val xor = a ^ b
val not = ~a
val add = a + b
val sub = a - b
val neg = -a
val mul = a * b
val div = a / b
val mod = a % b
val eq = a === b
val neq = a =/= b
```

Operator	Description	Data types
* / %	multiplication, division, modulus	UInt, SInt
+ -	addition, subtraction	UInt, SInt
=== =/=	equal, not equal	UInt, SInt, returns Bool
> >= < <=	comparison	UInt, SInt, returns Bool
<< >>	shift left, shift right (sign extend on SInt)	UInt, SInt
~	NOT	UInt, SInt, Bool
& | ^	AND, OR, XOR	UInt, SInt, Bool
!	logical NOT	Bool
&& ||	logical AND, OR	Bool

Bit extraction and concatenation

```
val sign = x(31)
val lowByte = largeWord(7, 0)
val word = highByte ## lowByte
val word = Cat(highByte, lowByte)
```

Scala expressions

`if`:

```
val res = if(x > y) x-y else y-x
```

`while`:

```
while(flag) {
    println("hello")
}
do {
    println("hello")
} while(flag)
```

`for`:

```
for(i <- 1 to 5) {
    println("hello")
}
for(i <- 1 until 6) {
    println("hello")
}
val arr = Array(1, 2, 3, 4, 5)
for(i <- arr) {
    println("hello")
}
```

Chisel expressions

`when` : (for hardware design)

```
when(cond 1) {def 1}
.elsewhen(cond 2) {def 2}
.otherwise {def default}
```

Bundle

Similar to `struct`

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

Vec

Similar to arrays

```
Vec(3, UInt(4.W))
```

Hardware

`Wire` : combinational logic

`Reg` : register (collection of D flip-flops)

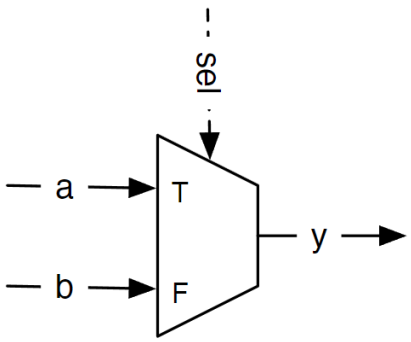
`IO` : a connection of a module

```
// signal
val w = Wire(UInt())
w := a & b
val w1 = WireDefault(5.U(8.W))

val ch = Wire(new Channel())
ch.data := 123.U
ch.valid := true.B
val b = ch.valid
val channel = ch

val v = Wire(Vec(3, UInt(4.W)))
v(0) := 1.U
v(1) := 3.U
v(2) := 5.U
val index = 1.U(2.W)
val a = v(index)

// register
```

```scala
val reg = RegInit(0.U(8.W))
reg := d
val q = reg

val nextReg = RegNext(d) // connected to input d
val bothReg = RegNext(d, 0.U) // connected to input d and reset to 0

val regVec = Reg(Vec(3, UInt(8.W)))
val dout = regVec(rdIdx)
regVec(wrIdx) := din

val initReg = RegInit(VecInit(0.U(3.W), 1.U, 2.U))
val resetVal = initReg(sel)
initReg(0) := d
initReg(1) := e
initReg(2) := f
```

Multiplexer

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



Counting

```scala
val cntReg = RegInit(0.U(8.W))
cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```