实验三：LC3 系统-ALU 模块设计

1. 实验目的

1.1. 熟悉 LC3 微结构。
1.2. 学习设计 ALU 的前置 Chisel 语法。
1.3. 掌握 LC3 系统中 ALU 模块的设计。
1.4. 掌握使用单元测试验证 LC3 系统中的 ALU 模块。

2. 实验内容

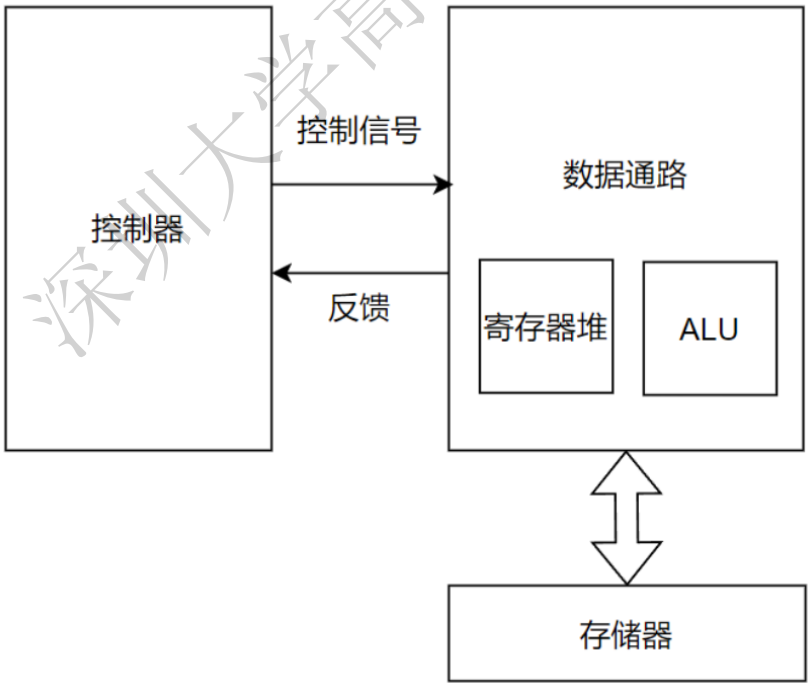
2.1. 阅读 《计算机系统概论》 附录 C，了解 LC3 微结构。
2.2. 阅读本实验前置 Chisel 语法。
2.3. 根据实验指导设计 ALU 模块。
2.4. 根据实验指导对 ALU 模块进行单元测试。

3. 实验步骤

3.1. 学习 LC3 结构

《计算机系统概论》 附录 C 中的内容为 LC3 内部架构的细节和规范，相当于设计的图纸，熟悉内部架构对后续实验有极大帮助。

在完成前两章实验，相信大家对基本的硬件设计概念与 Chisel 语言有所了解，实验三至七将会根据模块自第向上完成 LC3 的设计。 在整体实验前我们需要熟悉 LC3 的整体架构，从模块上看 LC3 主要分为三个大模块，即控制器、数据通路和存储器，其中数据通路中包含两个较为重要的模块，ALU 和寄存器堆，结构如下图所示。



三个大模块的工作流程为控制器发送控制信号到数据通路，数据通路根据信号进行取指、运算、访存等操作，其中访存操作数据通路会读写存储器。完成操作后数据通路将一些结果信号返回给控制器，控制器依据结果转换内部状态机，

生成新的控制信号，并重复“控制器发送控制信号——数据通路进行操作并返回结果——根据结果发送新控制信号”的流程。

根据以上架构，按照模块设计将分为五个实验，内容如下：

- 实验三：实现 LC3 的 ALU 模块，并对其进行单元测试。
- 实验四：实现 LC3 的寄存器堆模块，并对其进行单元测试。
- 实验五：实现 LC3 的控制器堆模块，并对其进行单元测试。
- 实验六：实现数据通路中的关键电路的设计。
- 实验七：了解 LC3 存储器的设计。

重要勘误

注意：如果你使用的是虚拟机，后续实验的代码放置在 ~/Frontend/chisel_lc3 文件夹中

实验开始前，请在 chisel_lc3 文件夹的 scala 文件全部添加上一个.bak 的后缀，否则使用 mill 运行仿真的时候会有各种报错，这是因为实验代码将部分内容留空给学生进行填空，这会导致 mill 的编译提示语法错误，如下图所示：

```
lc3@lc3-virtual-machine:~/Frontend/chisel_lc3$ mill chisel_lc3.test.testOnly example.WireTest
[37/71] chisel_lc3.compile
[info] compiling 9 Scala sources to /home/lc3/Frontend/chisel_lc3/out/chisel_lc3/compile.dest/classes ...
[error] /home/lc3/Frontend/chisel_lc3/src/main/scala/LC3/DataPath.scala:150:3: illegal start of simple expression
[error]   val dstData = WireInit(regfile.io.wData)
[error]   ^
[error] one error found
1 targets failed
chisel_lc3.compile Compilation failed
```

统一添加后缀的命令方式如下：

```
cd src
find . -name "*.scala" -exec bash -c 'mv "$0" "$0.bak"' {} \;
```

如果你想全部去除这个后缀，命令如下：

```
cd src
find . -name "*.scala.bak" -exec bash -c 'mv "$0" "${0%.bak}"' {} \;
```

添加后缀之后，需要根据实验使用到的文件将文件的.bak 后缀去掉，例如下面的实验中使用到了 src/test/scala/example/Wire.scala 这个文件，此时已经被重命名为了 src/test/scala/example/Wire.scala.bak，那么我们需要将其重新命名回来：

```
mv src/test/scala/example/Wire.scala.bak src/test/scala/example/Wire.scala
```

接下来就能正常使用了：

```
mill chisel_lc3.test.testOnly example.WireTest
```

在接下来的实验中，你都需要执行类似的操作来将一些需要使用到的文件的.bak 后缀移除，并且如果 mill 命令已经能正常执行就不需要将这个后缀添加回去！

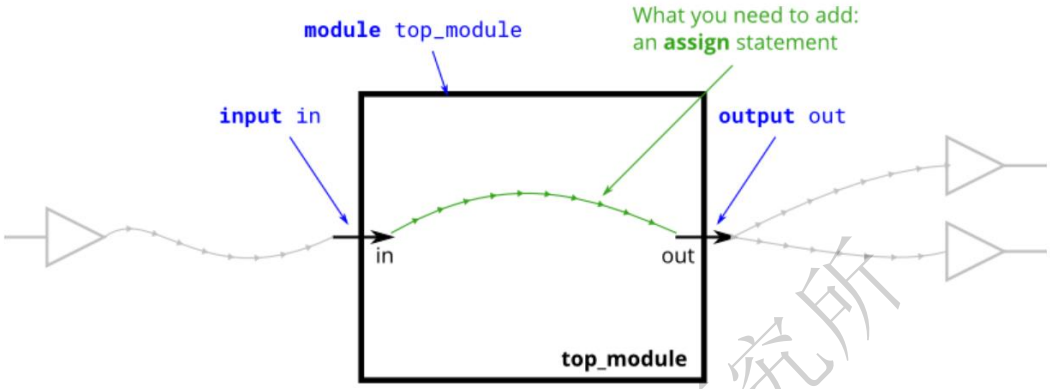
在接下来的实验中，你都需要执行类似的操作来将一些需要使用到的文件的.bak 后缀移除，并且如果 mill 命令已经能正常执行就不需要将这个后缀添加回去！

在接下来的实验中，你都需要执行类似的操作来将一些需要使用到的文件的.bak 后缀移除，并且如果 mill 命令已经能正常执行就不需要将这个后缀添加回去！

3.2. Chisel 语法学习

（1）Wire

电路中最基本的组件就是线，在 Chisel 中定义为 Wire。下图展示了一个只有一条线的模块，该模块只有一个输入端口 in，一个输出端口 out，in 端口信号直接通过线连接到 out。



在 Chisel 中一个线的定义为 val aWire = Wire(UInt(1.W))，其中 aWire 是线的名称，Wire 表示这是一个线类型，UInt 表示用整数表述信号，1.W 表示该线宽度为 1 比特。

完成线的定义，还需要把线连接起来，在 chisel 中连线操作符为 :=，如上图若定义绿色线为 aWire，则其连线为 aWire := io.in 和 io.out := aWire。当然，也可以直接 io.out := io.in，其结果是一样的。

下列代码（src/test/scala/example/Wire.scala）定义了上图模块和简单测试：

```scala
// 示例Wire模块
class Wire extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(1.W))
    val out = Output(UInt(1.W))
  })

  val aWire = Wire(UInt(1.W)) // 线定义

  aWire := io.in              // 线连接
  io.out := aWire

}
```

```scala
// 示例测试
class WireTest extends AnyFlatSpec with ChiselScalatestTester {

  it should "test wire" in {
    test(new Wire) { c =>
      c.io.in.poke(0.U)
      println(s"输入：${c.io.in.peek}, 输出：${c.io.out.peek}")

      c.io.in.poke(1.U)
      println(s"输入：${c.io.in.peek}, 输出：${c.io.out.peek}")
    }
  }

}
```

可以在项目根目录键入 `mill chisel_lc3.test.testOnly example.WireTest` 命令进行测试，其结果如下，输入信号为 0 时，输出也为 0，输入为 1 时输出也为 1。

```
$ mill chisel_lc3.test.testOnly example.WireTest
[71/71] chisel_lc3.test.testOnly
WireTest:
输入：UInt<1>(0), 输出：UInt<1>(0)
输入：UInt<1>(1), 输出：UInt<1>(1)
- should test switch
```

（2）简单逻辑操作符

电路中常用的运算有与、或、非、异或、加法等，运算符分别为 & | ~ ^ +
具体测试代码（src/test/scala/example/Logic.scala）如下：

```scala
// 示例Logic模块
class Logic extends Module {
  val io = IO(new Bundle {
    val ina  = Input(UInt(2.W))
    val inb  = Input(UInt(2.W))
    val add  = Output(UInt(2.W))
    val and  = Output(UInt(2.W))
    val or   = Output(UInt(2.W))
    val not  = Output(UInt(2.W))
    val xor  = Output(UInt(2.W))
  })

  io.add := io.ina + io.inb
  io.and := io.ina & io.inb
  io.or  := io.ina | io.inb
  io.not := ~io.ina
  io.xor := io.ina ^ io.inb

}
```

```scala
// 示例测试
class LogicTest extends AnyFlatSpec with ChiselScalatestTester {

  it should "test logic" in {
    test(new Logic) { c =>
      c.io.ina.poke("b01".U)
      c.io.ina.poke("b10".U)
      println(s"输入ina: ${c.io.ina.peek}, 输入inb: ${c.io.inb.peek}")
      println(s"输出add: ${c.io.add.peek}")
      println(s"输出and: ${c.io.and.peek}")
      println(s"输出or : ${c.io.or.peek}")
      println(s"输出not: ${c.io.not.peek}")
      println(s"输出xor: ${c.io.xor.peek}")
    }
  }

}
```

模块有两个输入端口，五个输出端口，均为 2bits。五个输出端口分别对输入端口做加、与、或、非、异或操作。测试模块对 ina 端口输入 01，对 inb 端口输入 10，检测五个输出端口数值。
键入测试命令得到结果：`mill chisel_lc3.test.testOnly example.WireTest`

```
LogicTest:
输入ina: UInt<2>(2), 输入inb: UInt<2>(0)
输出add: UInt<2>(2)
输出and: UInt<2>(0)
输出or : UInt<2>(2)
输出not: UInt<2>(1)
输出xor: UInt<2>(2)
- should test logic
```

（3）Switch

除了基本的与或非逻辑，在电路中还有一种选择逻辑，在 Chisel 中常用 Mux、When、Switch 来生成选择逻辑，本段介绍 Switch，具体语法为：

```
Switch(pattern) {
    is(match1) { logic1 }
    is(match2) { logic2 }
    ...
}
```

当 pattern 符合 match1 时，就会选择生成 logic1，当 pattern 符合 match2 时会选择生成 logic2。

示例代码（src/test/scala/example/Switch.scala）模块如下：

```scala
// 示例Switch模块
class Switch extends Module {
  val io = IO(new Bundle {
    val ina     = Input(UInt(4.W))
    val inb     = Input(UInt(4.W))
    val select  = Input(UInt(1.W))
    val result  = Output(UInt(4.W))
  })

  io.result := 0.U

  switch (io.select) {
    is(0.U) {io.result := io.ina}
    is(1.U) {io.result := io.inb}
  }
}
```

```scala
// 示例测试
class SwitchTest extends AnyFlatSpec
  with ChiselScalatestTester {

  it should "test switch" in {
    test(new Switch) { c =>
      c.io.ina.poke(3.U)
      c.io.inb.poke(6.U)
      c.io.select.poke(0.U)
      println(s"select为${c.io.select.peek}, " +
              s"result为${c.io.result.peek}")

      c.io.select.poke(1.U)
      println(s"select为${c.io.select.peek}, " +
              s"result为${c.io.result.peek}")
    }
  }
}
```

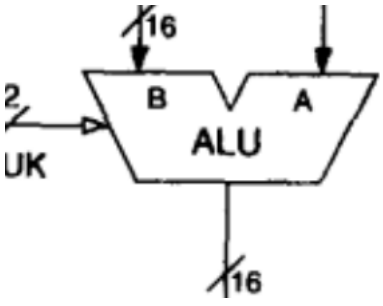
输入两个四位数据，一个一位选择信号，输出一个四位数据。该模块使用 Switch 实现选择功能，当 select 信号为 0 时，result 选择 ina 信号，否则选 inb 信号。在测试中 ina 输入 3.U,inb 输入 6.U，select 分别输入 0.U 和 1.U 观察 result 结果。

键入 `mill chisel_lc3.test.testOnly example.WireTest` 得到测试结果如下：

```
[71/71] chisel_lc3.test.testOnly
SwitchTest:
select为UInt<1>(0), result为UInt<4>(3)
select为UInt<1>(1), result为UInt<4>(6)
- should test switch
```

以上介绍了部分组合电路的语法，更多语法简介见文末附录 Chisel Cheat Sheet。

3.3. ALU 模块的设计



LC3 ALU 模块中需要接收三个输入，即操作数 1，操作数 2，和操作类型，并且得到一个输出。LC3 ALU 中支持的运算操作共四类，加法、按位或、按位反以及取操作数 1。在 ALU 运算时会根据上述的输入类型进行选择做何种运算（2bits 的操作类型可表示四种操作），其中操作类型的 2bits 来自于控制器的 SIG_ALUK 信号。因此其模块端口和操作类型如下表所示：

端口名称	方向	位宽
操作数1	输入	16 bits
操作数2	输入	16 bits
操作类型	输入	2 bits
输出数据	输出	16 bits

操作类型	功能
SIG_ALUK = 0	加法
SIG_ALUK = 1	按位与
SIG_ALUK = 2	对操作数1取反
SIG_ALUK = 3	取操作数1

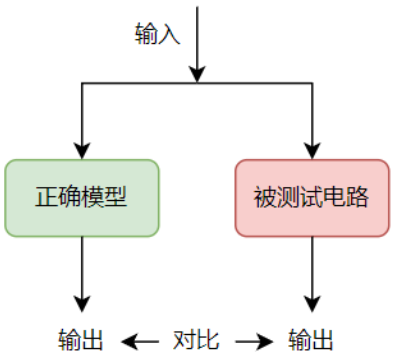
在有了上述端口和功能描述后，将其用 Chisel 代码描述。
（src/main/scala/LC3/ALU.scala）

```
class ALU extends Module{
  val io = IO(new Bundle{
    val ina = Input(UInt(16.W))
    val inb = Input(UInt(16.W))
    val op  = Input(UInt(2.W))     //ADD,AND,NOT,PASSA
    val out = Output(UInt(16.W))
  })

  io.out := DontCare

  // 实验三 任务一
  // 在此编写ALU逻辑

}
```

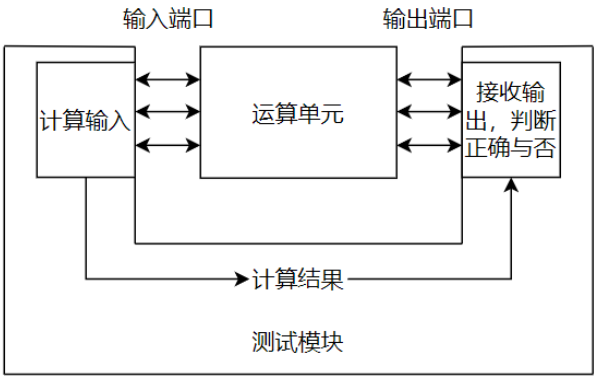
实验三-任务一：在 ALU.scala 文件中编写 ALU 逻辑（提示，可使用上述介绍语法描述该电路）

3.4. ALU 模块的单元测试

完成设计之后，我们需要对 ALU 的功能进行验证。除了在第三章中使用简单的结果观察方法(容易出错)，在数字电路中的验证方法通常采用一个正确的模型(golden modle)去和我们的设计进行对比。 给正确模型和被测试的电路给同样的输入，判断输出是否正确。



在 chisel 设计中最容易获得的模型便是使用 scala 编写的，从另一个角度看第三章单元测试的原理图，其结构大致相同，只是模块的输入输出不再是自己定义而是由程序自己计算好。 原理如下图所示。



ALU 测试代码（src/test/scala/ALUTest.scala）如下，其中本测试用例给出 golden module 的生成和 add 的功能测试，输入如下命令对 ALU 进行仿真验证。

```
mill chisel_lc3.test.testOnly LC3.ALUTest
```

```scala
class ALUTest extends AnyFlatSpec with ChiselScalatestTester {

  def TEST_SIZE = 10

  val ina, inb, add_out, and_out, not_out, pass_out = Array.fill(TEST_SIZE)(0)

  // 生成正确模型结果
  for (i <- 0 until TEST_SIZE) {
    ina(i) = Random.nextInt(0xffff)
    inb(i) = Random.nextInt(0xffff)
    add_out(i) = ina(i) + inb(i)
    and_out(i) = ina(i) & inb(i)
    not_out(i) = 0xffff-ina(i)
    pass_out(i) = ina(i)
  }

  // 硬件部分
  it should "test add" in {
    test(new ALU) { c =>
      for(i <- 0 until TEST_SIZE) {
        c.io.ina.poke(ina(i).U)
        c.io.inb.poke(inb(i).U)
        c.io.out.expect(add_out(i).U(15,0))
        println(s"${i}. ina=${ina(i)}  inb=${inb(i)}")
        println(s"${i}. 标准结果 add_out=${add_out(i) % (1<<16)} 模块结果 io.out=${c.io.out.peek}")
      }
    }
  }
```

输出结果为：

```
ALUTest:
0. ina=29156  inb=58051
0. 标准结果 add_out=21671 模块结果 io.out=UInt<16>(21671)
1. ina=17457  inb=50141
1. 标准结果 add_out=2062 模块结果 io.out=UInt<16>(2062)
2. ina=59201  inb=34146
2. 标准结果 add_out=27811 模块结果 io.out=UInt<16>(27811)
3. ina=33054  inb=2740
3. 标准结果 add_out=35794 模块结果 io.out=UInt<16>(35794)
4. ina=36058  inb=58677
4. 标准结果 add_out=29199 模块结果 io.out=UInt<16>(29199)
5. ina=734  inb=420
5. 标准结果 add_out=1154 模块结果 io.out=UInt<16>(1154)
6. ina=55545  inb=40688
6. 标准结果 add_out=30697 模块结果 io.out=UInt<16>(30697)
7. ina=53167  inb=2985
7. 标准结果 add_out=56152 模块结果 io.out=UInt<16>(56152)
8. ina=64668  inb=19943
8. 标准结果 add_out=19075 模块结果 io.out=UInt<16>(19075)
9. ina=49390  inb=31438
9. 标准结果 add_out=15292 模块结果 io.out=UInt<16>(15292)
- should test add
```

实验三-任务二：模仿上述 add 测试用例，在 ALUTest.scala 文件中编写 and, not, pass 功能的测试用例。

附录：

Chisel3 Cheat Sheet

Version 0.5.3 (beta): January 17, 2022

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`	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
`RegNext(next, init)`	update, with 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`

```
my_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`	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 := true.B
```

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: UInt / idx: Int)
myVec(ind: UInt / 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`

```
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:

```
Counter(n:Int): UInt
```

.inc() bumps counter returning true when `n` reached
.value returns current value

```
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