实验二：基本的组合逻辑电路

1. 实验目的

1.1. 学习 Chisel 的硬件类型和连线，包括 chisel 的赋值、端口（IO）、模块（Module）和线网（Wire）；

1.2. 学习 Chisel 的主要数据类型（UInt 和 Bool 类型）以及硬件中信号数据传递的概念，能够使用 Chisel 实现基础的组合逻辑电路；

1.3. 学习选择器 Mux 的使用，理解多路选择器的基本原理和 chisel 的实现方法。

2. 实验内容

2.1. 学习构建 Chisel 工程，完成 chisel-template 工程的构建和运行；

2.2. 通过修改 chisel-template 工程，编译生成代码；

2.3. 使用 Chisel 完成一个简单的路由器作业。

3. 实验步骤

3.1 从一个简单的例子讲起

3.1.1 逻辑和表达式化简

选择器（multiplexer）是一种最常用的组合逻辑电路。 根据选择（select)信号的值它从多个可能的输入中选择一个作为输出。 选择器有时简称为 mux。选择器的基本工作原理是通过选择信号来控制输出信号的来源，即根据选择信号的不同组合，将输入信号中的一个传递到输出端。

图 1 为 2-1 选择器电路符号，D0 和 D1 为输入信号，S 为选择信号，Y 为最终的输出信号。

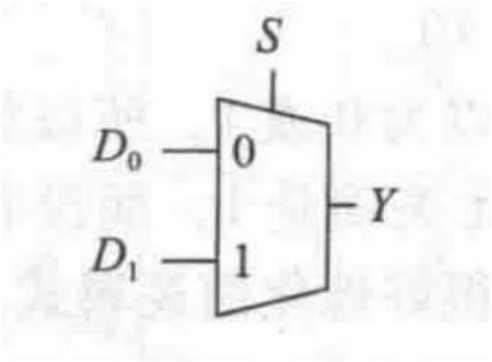


图 1 2-1 选择器抽象电路图

下表为为 2-1 选择器电路真值表：

S	D0	D1	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

使用传统的数字电路设计方法，我们会先使用卡诺图化简表达式，如图 2 所示：

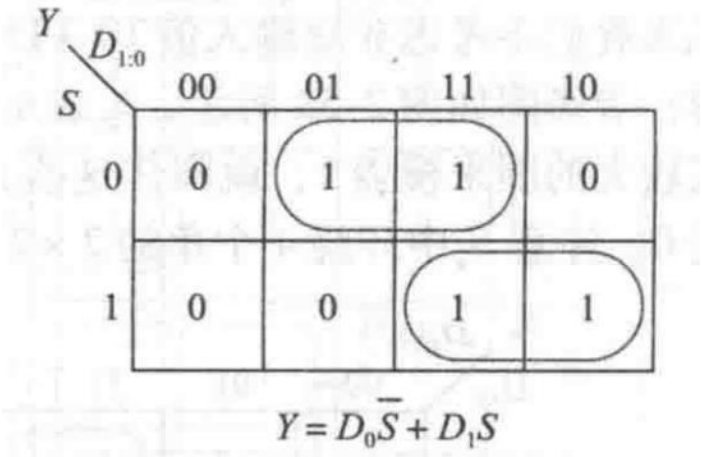


图 2 复用器卡诺图化简

最终得到门级电路图：

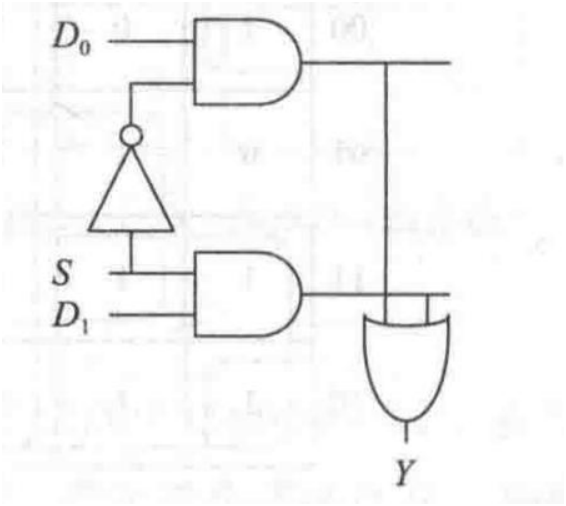


图 3　2-1 选择器门级电路

图 3 是一个使用与门、或门和非门实现的复用器。图 1 是电路的抽象表达形式。

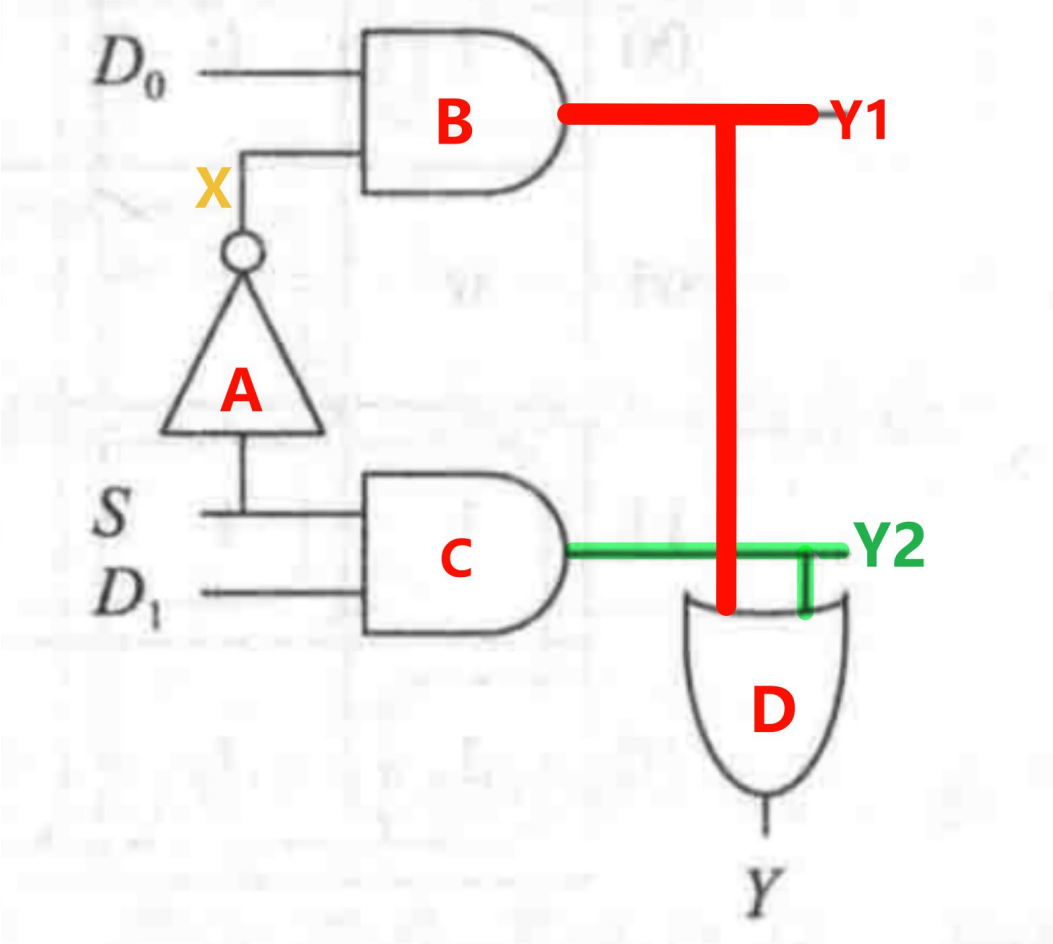


图 4 2-1 选择器门级电路分解图

图 2 中利用卡诺图化简的表达式可以分解为图 4。图 4 中共有 4 个逻辑门 A、B、C 和 D，其中 A 是非门，B 和 C 是与门，D 是或门。图中每一条线都称为同一个线网（wire），例如图中绿色部分为 Y2 线，图中红色为 Y1 线。表达式可以分解为 4 个部分：

1. 信号 S 经过逻辑门 A（非门）变为 X，则 $X = \overline{S}$

2. S 信号和 D1 信号经过逻辑门 C（与门）变为 Y2，则 Y2=S & D1

3. 同理，X 信号经过逻辑门 B（与门）变为 Y1，则 Y1=X & D0，带入 1 中的 X，得到 Y1=S & D0

4. Y1 信号和 Y2 信号经过逻辑门 D（或门），得到信号 Y，即 Y=Y1 | Y2

5. 综上可以得到 $Y = D0 \& \overline{S} | D1 \& S$

3.1.2 端口声明

不论我们使用 verilog 还是 chisel 描述电路，我们都是从端口开始描述。从外部来看，一个模块有输入和输出的信号，看不到内部的实现逻辑。在本次 2-1 选择器的例子中，如图 1 所示，从外部来看，输入信号只有 D0、D1 和 S，选择信号有 Y。

对于 Verilog 来说，首先描述端口信号，并且声明端口信号的方向：

```
1.  module mux(
2.    input  S,
3.           D0,
4.           D1,
5.    output Y
6.  );
7.
8.  //内部逻辑实现
9.  endmodule
```

第 1 行定义一个名为 mux 的 module，并且声明 4 个端口信号 S、D0、D1 和 Y，同时指定这些信号的方向输入信号在前面记 input，输出信号在前面记 output。这些信号名与图 4 中的标注一一对应。在模块的最后需要以 endmodule 结尾表示模块的定义结束。

我们可以使用 Chisel 来实现这一模块，首先在代码的开头需要导入 chisel3 中的库：

```
1.  import chisel3._
```

这句话表示将 chisel3 这个包中的所有方法都导入进来，这是我们所有代码运行都需要的包，后续如果没有特殊说明，都默认导入这个包，不在实际的代码中展示。

下面是实际的模块定义：

```
1.  class mux extends Module{
2.    val io = IO(new Bundle(){
3.      val S = Input(Bool())
```

```
    val D0 = Input(Bool())
    val D1 = Input(Bool())
    val Y = Output(Bool())
  })
//内部模块连线
}
```

接下来我们将逐行解释这些代码的作用与含义，并与 Verilog 代码和实际的电路对应起来。

1. 第 1 行代码定义了一个名为 mux 的 class，这个 class 继承于 Module 模块类。

Chisel 中还有其他 module 可以继承，我们最常用的是 Module 类。Module 类有一个隐式时钟（称为 clock）和一个隐式复位（称为 reset），必须实现抽象成员 io。在本实验课程中，只需要记住每次定义模块时都从 Module 继承即可，并且一定要实现 io 这一成员。代码模板如下：

```
class module_name extends Module {
  val io = IO(new Bundle {
    // define input/output ports
  })
  // module body
}
```

它等价于 Verilog 中的：

```
module module_name(port_list);
  // module body
endmodule
```

2. 第 2 行开始实现 io 这一成员。

定义一个模块前首先需要定义好端口。整个端口列表是由方法 IO[T<: Data](iodef: T) 来定义的，其参数通常是一个 Bundle 类型的对象（Bundle 类型大家可以理解为 chisel 的结构体类型），而且引用的字段名称必须是“io”(继承自 Module 的模块)。

因为端口存在方向，所以还需要方法 Input[T<: Data](source: T) 和 Output[T<: Data](source: T) 来为每个端口都表明具体的方向。注意，Input[T<:Data](sourceT) 和 Output[T<: Data](source: T) 的入参是数据类型，不能是硬件类型。

每个端口信号都有不同的数据类型，从第 3 行到第 6 行中一共出现了 2 种数据类型，分别是：Bool 类型和 UInt 类型。Bool 类型和 UInt 类型在电路上都表示比特信号。UInt 类型可以是任意宽度，而 Bool 类型是 UInt 类型的特殊形式，表示 1 bit 的 UInt 类型。这与 Verilog 中的 Wire 是一致的。关于 Chisel 的数据类型可以查看 3.2 小节。这里先介绍快速入门的方法：

在定义端口时，可以使用以下模板：

```
class MyModule extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  // 在这里实现模块的功能
}
```

在 io 这一成员变量中声明端口的名字（in1 或 in2），声明端口的方向（Input 或 Output），声明端口的数据类型（Bool 或者 UInt），如果是 UInt，需要声明宽度，若宽度为 x，则记为 UInt(x.W)。其中 W 为 width 的缩写（关于宽度和变量类型的定义会在后续详细介绍）。

3.1.3 内部逻辑实现

在图 2 我们化简得到了 2:1 选择器的逻辑表达式，图 4 我们也得到了 2:1 选择器的门级电路图。接下来我们来完成内部的逻辑实现。

对于 verilog 而言，可以直接根据逻辑表达式写出最终的输出结果：

```
module mux(
  input  S,
         D0,
         D1,
  output Y
);

  wire X = ~S;
  wire Y1 = D0 & X;
  wire Y2 = D1 & X;
  assign Y = Y1 | Y2;
endmodule
```

在 chisel 中类似于图 4 中的 X、Y1 和 Y2 信号，我们用 Wire 类型来定义。

接下来我们用 chisel 来描述：

```
class mux extends Module{
  val io = IO(new Bundle(){
    val S = Input(Bool())
    val D0 = Input(Bool())
    val D1 = Input(Bool())
    val Y = Output(Bool())
  })
  val X = Wire(Bool())//先定义 Wire 类型，再用:=赋值
  X := !io.S
```

```
10.  val Y1 = io.D0 & X //定义时直接赋值，注意第一次用=，第二次用:=
11.  val Y2 = io.D1 & X
12.  io.Y := Y1 | Y2
13. }
```

从第 8 行到第 13 行为内部的逻辑实现，这里用到了 3 种不同的 chisel 运算符，这里与 Verilog 的描述方法没有太大区别。主要是使用了一些运算符号。用到的 Chisel 逻辑运算符和对应的 Verilog 如下所示。关于 Chisel 运算类型，更详细的请看 3.2.4 小节运算类型。

Chisel	Explanation	width	in Verilog
!x	逻辑非	1	~ signal_x
x & y	位与	max(w(x), w(y))	signal_x & signal_y
x | y	位或	max(w(x),w(y))	signal_x | signal_y

由于在第 2 行定义了 io 这一 IO 结构体，则内部的 S、D0 等皆为 io 变量的成员，因此后续在对端口信号进行操作时，均以 io 为开头后加 . 的形式进行调用，例如 io.S、io.D0。

因为 chisel 基于 scala，scala 是一种强类型的语言，因此在声明变量的时候要指定变量的类型，例如第 8 行，我们声明中间变量 X 是一个 Wire 类型。wire 类型表示硬件单元之间的物理连线，由其连接的器件输出端连续驱动。

例如，图 4 表示将 in 的连接到 out，out 的值由 in 决定。具体来说，当 in 的值发生变化时，out 将立即反映这种变化，这种直接的连线关系确保了数据在器件之间的传输是实时的。

图 4 wire 连线示意图

在第 9 行中，除了声明 Y1 是 Wire 类型，还需要说明 Wire 的数据类型是 Bool 还是 UInt（本学期实验用到的两个类型），其中 Bool 类型是位宽为 1 的 UInt 类型。在 chisel 中，Bool 类型在物理上表示位宽为 1 的比特信号，常常用于逻辑运算。而 UInt 类型通常会为其指定位宽，例如宽度为 x 的 UInt 变量我们会写为 UInt(x.W)，其中 W 为 Width 的缩写。关于 UInt 的具体使用方法会在后续进行介绍。

除了像第 8 行直接声明变量的类型，也可以像第 10 行一样，直接写出运算式，此时 chisel 将会自动推断 Y1 的类型为一个 Bool 类型。

如果先在第 8 行对 X 变量进行了类型声明，那么在接下来对 X 变量进行赋值时，则需要使

用:=符号表示电路的连接。第 9 行则表示把端口信号 io.S 的值进行取反后赋值给 X。

同理我们可以写出 Y1 和 Y2 的表达式，最终 Y1 和 Y2 进行逻辑或后得到 mux 的输出信号 Y 的值（第 12 行）。

以上便完成了一个 2 选 1 选择器的定义。在 Chisel 中有许多封装好的模块（或者说硬件原语，也就是构成各种基本逻辑电路的组件，就像搭积木时候的积木一样），我们并不需要从门级电路开始构建，可以直接调用 Mux 方法来完成一个 2 选 1 的 Mux。

3.1.4 常用的硬件原语

chisel 的一大优势就在于有许多常用的硬件模块已经封装成库的形式，当我们需要的时候可以直接调用。对于我们上述实现的 mux，在 chisel 中已经有定义好的模块。在 3.1.2 中我们导入了以下包：

```
1.  import chisel3._
```

我们常用的 Mux 这个函数也在 chisel3 这个包中，因此不需要额外导入其他包。

代码如下：

```
1.  class mux_2 extends Module{
2.    val io = IO(new Bundle(){
3.      val S = Input(Bool())
4.      val D0 = Input(Bool())
5.      val D1 = Input(Bool())
6.      val Y = Output(Bool())
7.    })
8.    io.Y := Mux(io.S,io.D0, io.D1)
9.  }
```

io.Y 的输出使用了一个选择器，选择器的详细介绍请看 3.3.5 选择器 MUX 小节。这里的选择信号是 io.S，如果 io.S 的值为 1，也就是高电平，那么将会选择 io.D0 作为 io.Y 的值。如果 io.S 的值为 0，也就是低电平，将会选择 io.D1 作为 io.Y 的值。

3.2. Chisel 主要数据类型介绍

3.2.1 数据类型

在 chisel 中常用的数据类型为 UInt、SInt 和 Bool。实际的电路中信号都表现为 1 位或多位比特，只能用整数表示，这与 Verilog 是一致的。要表示浮点数，本质还是用多比特来构建，而且要遵循 IEEE 的浮点标准。对于 UInt，可以表示电路中任意位宽的线网或寄存器。对于 SInt，在 Chisel 里会按补码解读，转换成 Verilog 后会使用系统函数$signed，这是可综合的。对于 Bool，

转换成 Verilog 后就是 1bit 的线网或寄存器。建议大家入门时用 Bool 来表示 1bit 宽度的信号，用 UInt 来表示多位信号，例如数据或地址总线信号。

要表示值，则必须有相应的字面量。Chisel 定义了一系列隐式类:fomBigIntToLiteral、fromtIntToLiteral、fromtLongToLiteral、fromStringToLiteral、fromBooleanToLiteral 等。

以隐式类 fromtlntToLiteral 为例，存在一个同名的隐式转换，把相应的 Scala 的 Int 对象转换成一个 fomtIntToLiteral 的对象。而 fromtIntToLiteral 类有两个方法 U 和 S,分别构造一个等值的 UInt 对象和 SInt 对象而且 Scala 的基本值类都用字面量构造对象，所以要表示一个 UIt 对象，可以写成“1.U”的格式，这样编译器会插入隐式转换，变成“fomtIntToLiteral(1),U”，进而构造出字面值为 1 的 UInt 对象。同理，也可以构造 SInt。还有相同行为的方法 asUInt 和 asSInt。

从几个隐式类的名字就可以看出,可以通过 BigInt、Int、Long 和 String 这 4 种类型的 Scala 字面量来构造 UInt 和 SInt。按 Scala 的语法，其中 BigInt、Int、Long 三种类型默认是十进制的，但可以加前缀“0x”或“0X”变成十六进制的。

对于字符串类型的字面量，Chisel 编译器默认也是十进制的，但是可以加上首字母“h”“o”“b”来分别表示十六进制、八进制和二进制。此外，字符串类型的字面量可以用下画线间隔。

可以通过 Boolean 类型的字面量 true 和 false 来构造 fromBooleanToLiteral 类型的对象,然后调用名为 B 和 asBool 的方法进一步构造 Bo0l 类型的对象。例如:

```
1.U              // 字面值为"1"的 UInt 对象
0xd.U             // 字面值为"13"的 UInt 对象
-8.S             // 字面值为"-8"的 sInt 对象
"b01_01".U             // 字面值为"5"的 UInt 对象
true.B           // 字面值为"true"的 Bool 对象
```

3.2.2. 数据宽度

默认情况下，数据的宽度按字面值取最小，例如，字面值为“8”的 UInt 对象为 4 位宽，SInt 为 5 位宽，但是也可以指定宽度。

Chisel 专门设计了宽度类 Width。还有一个隐式类 fromIntToWidth，把 Int 对象转换成 fromIntToWidth 类型的对象，然后通过方法 W 返回一个 Width 对象。方法 U、asUInt、S 和 asSInt 都有一个重载的版本，接收一个 Width 类型的参数，构造指定宽度的 SInt 和 UInt 对象。注意，1.U (32)表示的是取 1.U 的第 32 位，Bool 类型固定为 1 位宽。例如：

```
1.U //字面值为"1、宽度为 1bit 的 UInt 对象
```

```
2.  1.U(32.W)//字面值为"1、宽度为 32bit 的 UInt 对象
3.  1.U(32)//字面值为"0"、宽度为 1bit 的 UInt 对象
4.  1.U(0)//字面值为"1"、宽度为 1bit 的 UInt 对象
```

工厂方法构造没有字面量的对象。UInt 和 SInt 的 apply 工厂方法有两个版本，一个版本接收 UInt、SInt 和 Bool 都不是抽象类，除可以通过字面量构造对象外，也可以直接通过 applyWidth 类型的参数构造指定宽度的对象，另一个则是无参版本构造位宽可自动推断的对象。有字面量的数据类型用于赋值、初始化寄存器等操作，而无字面量的数据类型则用于声明端口、构造向量等。

结合 3.2.1 小节的数据类型和 3.1 中讲过的赋值方法，我们就可以在模块内部进行类似下面的声明：

```
1.  val v0 = Wire(UInt(2.W))
2.  val v1 = Wire(UInt(4.W))
3.  val b0 = Wire(Bool())
4.  val b1 = Wire(Bool())
5.
6.  v0 := 2.U
7.  v1 := "b01_01".U
8.  b0 := 1.U
9.  b1 := true.B
```

前面我们提到，Bool 变量其实是位宽为 1 的 UInt 变量，因此第 8 和第 9 行将 Bool 变量赋值为 1.U 和 true.B 其实是等价的。

3.2.3. 类型转换

UInt、SInt 和 Bool 三种类型都包含 4 个方法:asUInt、asSInt、asBool 和 asBools。其中 asUnt 和 asSInt 分别把字面值按无符号数和有符号数解释，并且位宽不会变化，要注意转换过程中可能发生符号位和数值的变化。例如，3bit 的 UInt 值“b111”，其字面量是“7”，转换成 SInt 后字面量就变成了“-1”。asBool 会把 1bit 的“1”转换成 Bool 类型的 true，把“0”转换成 false。如果位宽超过 1bit，则用 asBools 转换成 Bool 类型的序列 Seq[Bool]。

```
1.  "ha".asUInt(8.w)//字面值为"0xa"、宽度为 8bit 的 UInt 对象
2.  1.S(3.W).asUInt//字面值为"1"、宽度为 3bit 的 UInt 对象
```

3.2.4 运算类型

逻辑运算

Chisel	Explanation	width
!x	逻辑非	1
x && y	逻辑与	1
x||y	逻辑或	1

位操作运算符

Chisel	Explanation	width	in Verilog
~x	位反	w(x)	~ signal_x
x & y	位与	max(w(x), w(y))	signal_x & signal_y
x | y	位或	max(w(x), w(y))	signal_x | signal_y
x ^ y	按位异或	max(w(x), w(y))	signal_x ^ signal_y
x(n)	bit 索引, 0 is LSB	1	signal[n]
x(n, m)	字段提取	n - m + 1	signal [n:m]
x << n	左移	w(x) + n	<<
x >> n	右移	w(x) - n	>> (补零)
Fill(n, x)	重复拼接	n * w(x)	{n{signal_x}}
Cat(x, y)	拼接	w(x) + w(y)	{signal_x,signal_y}
Mux(c, x, y)	三元操作	max(w(x), w(y))	signal_c ? signal_x : signal_y
v.andR	位与	1 (Bool)	& signal_v
v.orR	位或	1 (Bool)	| signal_v
v.xorR	位异或	1(Bool)	^ signal_v

关系运算符

Chisel	Explanation	width	in Verilog

Chisel	Explanation	width	in Verilog
x === y	相等(triple equals)	1	signal_x == signal_y
x != y	不等	1	signal_x != signal_y
x =/= y	不等	1	signal_x != signal_y
x > y	大于	1	signal_x > signal_y
x >= y	大于等于	1	signal_x >= signal_y
x < y	小于	1	signal_x < signal_y
x <= y	小于等于	1	signal_x <= signal_y

算数运算符

Chisel	Explanation	width	in Verilog
x + y	加	max(w(x),w(y))	signal_x + signal_y
x +& y	位扩加	max(w(x),w(y))+1	NULL
x - y	减	max(w(x),w(y))	signal_x - signal_y
x -& y	位扩减	max(w(x),w(y))+1	NULL
x * y	乘	w(x)+w(y)	signal_x * signal_y
x / y	除	w(x)	signal_x */ signal_y
x % y	取余	bits(maxVal(y)-1)	signal_x % signal_y

3.2.5 向量（Vec）和包裹（Bundle）类型

如果需要一个集合类型的数据，可以使用 Chisel 专属的 Vec 类型。Vec[T]，其中 T 是一种数据类型，比如可以是 UIn 或者 Bool，而且每个元素的类型、位宽必须一样。

Vec[T]的伴生对象里有一个 apply 工厂方法，接收两个参数:第一个是 Int 类型，表示元素的个数；第二个是元素。它属于可索引的序列，下标从 0 开始，可以使用 Int 类型索引，也可以使用 UInt 索引，例如下面代码：

```
1.  val myVec = Wire(Vec(3, UInt(2.W)))
```

```
2.    myVec(0) := 0.U
3.    myVec(1) := 1.U
4.    myVec(2) := 2.U
```

上述代码中，第一行声明了变量 myVec，硬件类型是 Wire，数据类型是 Vec，这个 Vec 可以看作是一个数组，Vec 的宽度为 3，可以视为有 3 个 UInt 元素，UInt 元素为的位宽为 2。需要使用 myVec 中的元素时，便可以使用下标进行索引，下标从 0 开始，因此可索引的下标为 0,1,2；每个元素的赋值和前面所说的没有区别。

引入了 Vec 以后，我们便可以简化我们的 mux 代码：

```
1.  class mux_3 extends Module{
2.    val io = IO(new Bundle(){
3.      val S = Input(Bool())
4.      val D = Input(Vec(2,Bool()))
5.      val Y = Output(Bool())
6.    })
7.    io.Y := Mux(io.S,io.D(0), io.D(1))
8.  }
```

第 4 行可以看到，对于输入信号 D0 和 D1，我们简化成了一个 Vec，Vec 的宽度为 2，元素类型为 Bool，在第 7 行引用变量时，可以通过 io.D(0)和 io.D(1)的形式。如果输入端口的数量有很多，可以简化 io 端口的声明，使得代码更加简洁。

对于 Bundle 类型，我们在前面定义 io 时已经使用过。Bundle 很像 C 语言的结构体，用户可以编写一个自定义类来继承它，然后在自定义的类里包含其他各种类型的字段。它可以协助构建线网或寄存器，其最常见的用途是构建一个模块的端口列表，或者一部分端口。

例如端口定义：

```
1.  class MyModule extends Module{
2.    val io = IO(new Bundle {
3.      val in = Input(UInt(32.W))
4.      val out = Output(UInt(32.W))
5.    })
6.  }
```

我们也可以把端口输入输出信号定义为一个 Bundle，例如 3.3.2 中的例子。

3.3. 主要硬件类型和连线

3.3.1 模块（Module）定义

在 Chisel 中有多种模块的类型，我们最常用的是 Module 类。Module 类有一个隐式时钟（称为 clock）和一个隐式复位（称为 reset），必须实现抽象成员 io。例如：

```
class AndModule extends Module{
    val io = IO(new Bundle {
        val a = Input(Bool)
        val b = Input(Bool)
        val c = Output(Bool)
    }
    io.c := io.a & io.b
}
```

在实现的抽象成员 io 内部声明所需的输入和输出端口，对于输入端口可以不使用，但对于输出端口则一定需要为其赋值，否则 Chisel 编译将无法通过。

3.3.2 端口（IO）

定义一个模块前首先需要定义好端口。整个端口列表是由方法 IO[T<: Data](iodef: T) 来定义的，其参数通常是一个 Bundle 类型的对象，而且引用的字段名称必须是“io”(对于继承自 Module 的模块)。因为端口存在方向，所以还需要方法 Input[T<: Data](source: T) 和 Output[T<: Data](source: T) 来为每个端口都表明具体的方向。注意，Input[T<:Data](sourceT) 和 Output[T<: Data](source: T) 的入参是数据类型，不能是硬件类型。

一旦端口列表定义完成，就可以通过 io.xxx 来使用。输入可以驱动内部其他信号，输出可以被其他信号驱动。可以直接进行赋值操作，Bool 类型的端口还能直接作为使能信号。端口不需要再使用其他硬件类型来定义，不过需要注意的是，从性质上来说它仍然属于组合逻辑的线网。例如:

```
class MyIO extends Bundle {
    val in=Input(Vec(5,UInt(32.W)))
    val out=Output(UInt(32.W))
}

class MyModule extends Module{
    val io=IO（new MyIo）// 模块的端口列表
}
```

对于两个相连的模块，可能存在大量同名但方向相反的端口。仅仅为了翻转方向而不得不重

写一遍端口，这样费时费力，所以 Chisel 提供了 Flipped[T<: Data](source: T) 方法，可以把参数里所有的输入端口转为输出端口,输出端口转为输入端口。例如：

```
class MyI0 extends Bundle{
    val in=Input(Vec(5，UInt(32.W)))
    val out = Output(UInt(32.W))
}

class MyModule_1 extends Module {
    val io=IO(new MyIO)// in 是输入，out 是输出
}

class MyModule_2 extends Module {
    val io=I0(Flipped(new MyIO))// out 是输入，in 是输出
}
```

3.3.3 硬件类型

模块内部主要有 Wire（线网）和 Reg（寄存器）两种数据类型，他们分别用于描述数字电路里的组合逻辑和时序逻辑。Wire 之间的连线是不具有周期延迟，而 Reg 的输出和输出之间有一周期的延迟时间。本节课我们仅使用到 Wire 类型，关于寄存器的使用将会在接下来的相关实验进行介绍。

3.3.4 赋值方法

有了硬件类型后，就可以用赋值操作来进行信号的传递或电路的连接了。只有硬件赋值才有意义，单纯的数据对象进行赋值并不会被编译器转换成实际的电路，因为在 Verilog 里也是对 wire、reg 类型的硬件进行赋值的。

在 Chisel 里，因为硬件电路具有不可变性，所有对象都应该由 val 类型的变量来引用。因此，一个变量一旦在初始化时绑定了一个对象，就不能再发生更改。但是，引用的对象很可能需要被重新赋值。例如，输出端口在定义时使用了“=”与端口变量名进行了绑定，等到驱动该端口时，就需要通过变量名来进行赋值操作，更新数据。很显然，此时“=”已经不可用了，因为变量在声明的时候不是 var 类型。即使是 var 类型，也只是让变量引用新的对象,而不是直接更新原来的可变对象。

为了解决这个问题，几乎所有的 Chisel 类都定义了方法“:=”，作为“=”赋值运算符的代替。所以首次创建变量时用“=”初始化，如果变量引用的对象不能立即确定状态或本身就是可变对象，则在后续更新状态时应该用“:=”。从前面讲的操作符优先级来判断，该操作符以“=”结尾，而且不是 4 种逻辑比较符号之一，所以优先级与“=”一致，是最低的,例如:

```
val x = Wire(UInt(4.W))
```

```
val y = Wire(UInt(4.W))
x := "b1010".U    //向 4bit 的线网 x 赋予了无符号数 10
y := ~x           //把 x 按位取反，传递给 y
```

3.3.5 选择器 MUX

选择器是一个很常用的电路模块，Chisel 内建了几种多路选择器。第一种形式是二输入多路选择器“Mux(sel,in1,in2)”，它在 chisel3 包中。sel 是 Bool 类型 in1 和 in2 的类型相同，都是 Data 的任意子类型。当 sel 为 true.B 时，返回 in1，否则返回 in2。

因为 Mux 仅把一个输入返回，所以 Mux 可以内嵌 Mux，构成 n 输入多路选择器。类似于嵌套的三元操作符，其形式为如下。

```
Mux(c1,a,Mux(c2,b, Mux(..., default)))
```

除了 Mux 以外，其他的选择器放在 chisel3.util 中，因此需要以下代码用来导入包：

```
import chisel3.util._
```

第二种多路选择器是 MuxCase，是针对上述 n 输入多路选择器的简便写法，形式为：

```
MuxCase(default, Array(c1 ->a, c2 ->b,.))
```

它的展开与嵌套的 Mux 是一样的。第一个参数是默认情况下返回的结果，第二个参数是一个数组，数组的元素是对偶“(成立条件,被选择的输入)”。MuxCase 在 chisel3.util 包中。

第三种多路选择器是 MuxLookup，是 MuxCase 的变体，它相当于把 MuxCase 的成立条件依次换成从 0 开始的索引值，就好像一个查找表，其形式为：

```
MuxLookup(idx,defaultArray(0.U -> a, 1.U -> b, ..)
```

它的展开相当于：

```
MuxCase(default, Array((idx === 0.U)-> a, (idx === 1.U)-> b, ...))
```

MuxLookup 也在 chisel3.util 包中。

第四种多路选择器是 Mux1H，是 chisel3.util 包中的独热码多路选择器，它的选择信号是一个独热码。如果零个或多个选择信号有效，则行为不确定。它有以下几种常用的定义形式代码引自 Chisel 3 官方源代码：

```
val hotValue = Mux1H(io.selector,Seq(2.U,4.U,8.U,11.U))

val hotValue = MuxlH(Seg(io.selector(0),io.selector(1),io.selector(2),io.selector(3)),Seq(
2.U,4.U,8.U,11.U))

val hotValue = MuxlH(Seq(
```

```
      io.selector(0)-> 2.U,
      io.selector(1)-> 4.U,
      io.selector(2)-> 8.U ,
      io.selector(3) -> 11.U
  ))
  //以上三种 Mux1H 定义形式是等价的，io.selector 是一个 UInt 类型的数据，并且位宽不能小于待选择数据的个数。在第一种形式中，Mux1H 会从低到高依次将 io.selector 的每一位作为一个选择信号，并和提供的被选择数据一一对应。
```

3.4. 基于 mill 的 chisel-template 工程的使用

由于 Chisel 是基于 Scala 实现的，实际上可以理解为 Chisel 就是 Scala 的一个库，因此 Chisel 的工程也是 Scala 的工程，而一个 Scala 工程通常会包括各种 package 的依赖，这些依赖需要由各种工程构建工具来管理，如：sbt、mill、gradle 等。Chisel 官方推荐用户使用 mill 来管理 Chisel 工程，mill 是一个现代化的 Scala 构建工具，提供了简洁高效的构建体验，与传统的 sbt 相比，mill 更加轻量和易于使用，它具有更快的启动时间和更清晰的配置语法。mill 支持多种编程语言和构建任务，但它在 Scala 项目中的应用尤为广泛。

Chisel 官方也提供了一个模板工程（chisel-template），旨在帮助新用户快速上手 Chisel 的开发。这个模板工程已经预先配置好了所需的依赖和基本的项目结构，使用户可以专注于编写 Chisel 代码，而不必花费时间在项目配置上。下面提供 chisel-template 的安装与使用步骤：

1）git clone 工程项目

如果使用的是实验提供的虚拟机镜像，可以前往 ~/Downloads 文件夹中复制 chisel-template 这个文件夹即可，不需要 git clone 项目。

```
git clone https://github.com/chipsalliance/chisel-template.git
```

在 clone 完成之后，我们需要将 chisel-template 切换到一个特定的提交，这是因为最新的 chisel-template 兼容的是新版本的 verilator，而虚拟机镜像使用的是旧版本的 verilator，使用旧版本的 chisel-template 可以消除这一兼容问题，同时不会影响我们教学实验的使用。

```
git checkout -f 6d298bd42c55c9d41fb588683a81760d33a97835
```

2）替换模板工程中的%NAME%字段

chisel-template 工程中有一个 %NAME% 字段用于标识这个 Chisel 工程的名字，我们需要替换掉这个字段才能正常运行 chisel-template。

```
# 切换到 chisel-template 工程目录
cd chisel-template

# 将 %NAME% 字段替换为 MyChiselProject(也可以是别的工程名字)
find . -type f -exec sed -i 's/%NAME%/MyChiselProject/g' {} +
```

3）安装 mill

mill 的安装很简单，只需要下面一条命令即可，如果你使用的是实验提供的虚拟机镜像，则可以跳过这一步，镜像中已经安装了 mill。

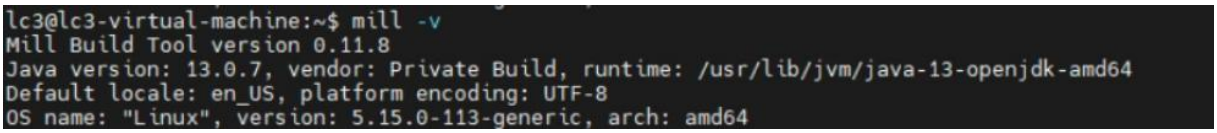
```
curl -L https://github.com/com-lihaoyi/mill/releases/download/0.11.8/0.11.8 > mill && chmod +x mill
```

4) 查看 mill 是否安装成功

如果是初次执行 mill 命令，会自动下载 mill 的相关文件，这可能会需要一段时间，也取决于网络环境。

```
mill -v
```

如果安装成功，将会获得如下的输出：



5) 尝试编译测试 chisel-template

```
cd <chisel-template>
mill MyChiselProject.test # 注意此处的 MyChiselProject 需要替换为你自己规定的工程名称，在这里工程名称就是 MyChiselProject
```

如果一切正常，将会得到如下的输出（期间可能会下载各种依赖，也有可能会因为网络问题而下载失败，请重试几次）：

```
lc3@lc3-virtual-machine:~/chisel-template$ mill MyChiselProject.test
[83/83] MyChiselProject.test.test
GCDSpec:
- Gcd should calculate proper greatest common denominator
```

6）chisel-template 项目结构介绍

chisel-template 使用成的 mill 来管理整个 Chisel（Scala） 工程，我们的源文件需要放在特定的位置才能被 mill 正确识别。

使用 tree 命令可以查看当前的目录结构，其中我们需要关心的是 src 文件夹的结构，src 文件夹放置的是整个工程中的源码以及测试代码，其中 src/main/scala 目录下 gcd 是我们工程的提供的一个 package，也是 chisel-template 提供的一个 demo 文件，gcd 是其 package 名，因此需要一个单独的文件夹，src/test/scala 目录同理，其中的 gcd 文件夹的目录是 gcd 这个 package 测试文件，我们执行 mill MyChiselProject.test 命令的时候执行的就是这里的测试文件：GCDSpec.scala。

```
lc3@lc3-virtual-machine:~/Downloads/chisel-template$ tree
.
├── build.sbt
├── build.sc
├── LICENSE
├── project
│   ├── build.properties
│   └── plugins.sbt
├── README.md
└── src
    ├── main
    │   └── scala
    │       └── gcd
    │           ├── DecoupledGCD.scala
    │           └── GCD.scala
    └── test
        └── scala
            └── gcd
                └── GCDSpec.scala
```

如果我们要创建一个新的 chisel 代码，可以在 src/main/scala 目录下，创建一个新的文件夹，比如 mymodule（注意，package 通常是小写的，因此 package 的目录名也是小写的），同时如果你需要写测试文件，可以在 src/test/scala 目录下创建一个类似的文件夹，例如：

```
lc3@lc3-virtual-machine:~/chisel-template$ tree
.
├── build.sbt
├── build.sc
├── LICENSE
├── project
│   ├── build.properties
│   └── plugins.sbt
├── README.md
└── src
    ├── main
    │   └── scala
    │       ├── gcd
    │       │   ├── DecoupledGCD.scala
    │       │   └── GCD.scala
    │       └── mymodule
    │           └── MyModule.scala
    └── test
        └── scala
            ├── gcd
            │   └── GCDSpec.scala
            └── mymodule
                └── TestMyModule.scala
```

这样我们就在 chisel-template 里添加上了我们自己的工程文件了，接下来的实验中如果需要创建工程文件，也是按照这个思路来创建。

3.5 常见的错误

3.5.1 对 val 定义的变量重复赋值

```
1.  val Y1 = Wire(Bool())
2.  Y1 = io.D0 & X
```

在前面的例子中，如果对 Y1 赋值不是用:=将会报以下错误：

```
reassignment to val
  Y1 = io.D0 & X
```

这是因为第一行已经声明了 Y1 是一个 Wire 硬件类型，是不可变的，而后续对 Wire 的赋值都只能通过:=进行电路信号的连接。

3.5.2 重复赋值

```
1.  val Y1 = Wire(Bool())
2.  Y1 := true.B
3.  Y1 := false.B
```

在上述代码中第 2 和第 3 行都对 Y1 信号进行了赋值，最终将会是第 3 行的代码覆盖了第 2 行，从而 Y1 的值为 false.B，这种行为对 chisel 来说不会报错，但却常常违背我们的意图。例如在一串较长的代码中，前后距离较远，对同一个变量进行赋值，就容易出现错误。

3.5.3 chisel 的数据类型和 scala 的数据类型不匹配

```
1.  val Y1 = Wire(UInt(1.W))
2.  Y1 := 1
```

如果出现以上代码编译将会报错：

```
type mismatch;
 found   : Int(1)
 required: chisel3.Data
  Y1 := 1
```

这是由于 1 是 Int 类型，而 Y1 是一个 UInt 类型，所有数字都视为 Int 类型，不能直接赋值给 UInt，需要记为 Y1 := 1.U，通过.U 转化成 UInt 类型才能赋值给 Y1。这也是初学者经常会犯的错误。

3.6 实验任务

查看 src/main/scala/exp2/目录下的 Router.scala 文件，这是接下来的作业，内容如下：

```
1.  // 定义了一个名为Router 的Chisel 模块,继承自Module 类
2.  class Router extends Module{
3.    // 定义了一个Bundle 类型的端口,名为io
4.    // 这个Bundle 包含以下3 个端口:
5.   val io = IO(new Bundle(){
6.    // 一个2 位宽的无符号整型输入端口,名为ctrlInfo
7.     val ctrlInfo = Input(UInt(2.W))
8.    // 一个3 个元素的向量(Vec),每个元素是一个5 位宽的无符号整型输入端口,名为in
9.     val in = Input(Vec(3, UInt(5.W)))
10.   // 一个3 个元素的向量(Vec),每个元素是一个5 位宽的无符号整型输出端口,名为out
11.    val out = Output(Vec(3, UInt(5.W)))
12.  })
13.
14.  io.out(0) := io.in(0)  // 将in 向量的第0 个元素连接到out 向量的第0 个元素,以下同理
15.  io.out(1) := io.in(1)
16.  io.out(2) := io.in(2)
```

这是一个简单的 Router 模块,它将 3 个 5 位宽的输入端口直接连接到 3 个 5 位宽的输出端口,没有任何复杂的功能。它还有一个 2 位宽的控制信息输入端口 io_ctrlInfo,但在这个代码中并没有使用它，作为作业在后续由学生完成。

后续的作业需要通过修改 15 到 16 行的代码通过测试。

任务一：复现上述实验过程（基于 mill 的 chisel-template 工程）。

任务二：实现一个 3 输入和 3 输出的路由器，根据下表中控制信号的不同，将指定输入端口的信号传递到指定的输出端口。

输入端口（三个端口）	控制信号（2 bit）	输出端口（三个端口）
a,b,c	00	a,c,b
a,b,c	01	b,a,c
a,b,c	10	b,c,a
a,b,c	11	c,b,a

- 在目录 src/main/scala 下已经有文件 router.scala，在空白处完成你的实验。

```
1.  class Router extends Module {
2.    val io = IO(new Bundle {
3.      val ctrlInfo = Input(UInt(2.W))
4.      val in = Input(Vec(3, UInt(5.W)))
5.      val out = Output(Vec(3, UInt(5.W)))
6.    })
7.
8.    //TODO: 在这里完成你的作业。。。
9.    io.out(0) := io.in(0)
10.   io.out(1) := io.in(1)
11.   io.out(2) := io.in(2)
12. }
```

- 注意，不要修改端口信号！
- 代码完成后，在目录下运行 mill MyChiselProject.test.testOnly exp2.TestRouter，如下图所示，则表示所有测试已经通过。

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp2.TestRouter
[50/83] MyChiselProject.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestRouter:
- Router should pass these test
```