实验三：编码器与译码器

1. 实验目的

1.1. 学习编码器和译码器的基本原理，掌握常见的编码器与译码器原理概念；

1.2. 学习编码器与译码器在 Chisel 中的使用；

1.3. 学习查找表的基本原理，并且能够使用 Chisel 实现。

2. 实验内容

2.1. 介绍编码器和译码器的基本原理，介绍常见的 OneHot 编码；

2.2. 学习 Chisel 中的各种有关编码器与译码器的硬件原语的使用；

2.3. 介绍查找表的基本原理，学习如何使用高阶函数自动生成序列值。

3. 实验相关 Chisel 语法介绍

在本小节，我们将介绍接下来实验过程中使用到的 Chisel 语法。

3.1. switch 语句

switch 语句在 Chisel 中用于多路分支选择，类似于其他编程语言中的 switch 或 case 结构。它根据一个选择信号的值执行不同的代码块。下面是一个 switch 语句的示例：

```
switch(state) {
  is(s0) {
    // 当 state 为 s0 时执行的代码
  }
  is(s1) {
    // 当 state 为 s1 时执行的代码
  }
  is(s2) {
    // 当 state 为 s2 时执行的代码
  }
}
```

在这个例子中，state 是一个选择信号，根据它的值执行不同的代码块。

3.2. MuxLookup

MuxLookup 函数在 Chisel 中用于实现多路选择器（Mux），它根据选择信号的值查找对应的输入信号，并将其传递到输出端。下面是一个 MuxLookup 的示例：

```
val result = MuxLookup(sel, default, Array(
  (0.U) -> in0,
  (1.U) -> in1,
  (2.U) -> in2,
  (3.U) -> in3
))
```

在这个例子中，sel 是选择信号，当 sel 为 0 时，result 输出 in0；当 sel 为 1 时，result 输出 in1，依此类推。如果 sel 不匹配任何一个指定值，result 输出默认值 default。

3.3.when

when 语句在 Chisel 中用于条件判断和执行，相当于其他编程语言中的 if 语句。它根据条件表达式的值执行相应的代码块。下面是一个 when 语句的示例：

```
when (cond) {
  // 当 cond 为 true 时执行的代码
} .elsewhen (otherCond) {
  // 当 otherCond 为 true 时执行的代码
} .otherwise {
  // 当上述条件都不满足时执行的代码
}
```

在这个例子中，cond 和 otherCond 是条件表达式，当 cond 为真时执行第一个代码块，当 otherCond 为真时执行第二个代码块，否则执行最后的代码块。

3.4.Vec/VecInit

在 Chisel 中，Vec 这一数据结构用于定义向量（数组）类型，可以存储多个相同类型的元素。Vec 常用于需要处理多组数据的场景，例如多输入多输出信号的处理。

Seq

Seq 是 Scala 标准库中的一个集合类，可以存储多个元素。Seq 是不可变的，即一旦创建，其内容不能改变。Seq 通常用于初始化 Vec，例如在 VecInit 中。

```
val mySeq = Seq(1.U, 2.U, 3.U, 4.U)
```

在这个例子中，mySeq 是一个不可变的序列，包含四个元素，每个元素都是 1 位宽的无符号整数。

Vec

我们可以直接使用 Vec 来创建一个包含多个元素的向量，并且可以通过索引访问和操作这些元素。

例如，定义一个包含 4 个 8 位无符号整数的 Vec：

```
val myVec = Wire(Vec(4, UInt(8.W)))
```

在这个例子中，myVec 是一个包含 4 个 8 位宽无符号整数的向量。我们可以通过索引访问和修改 Vec 中的元素：

```
myVec(0) := 10.U
val firstElement = myVec(0)
```

VecInit

VecInit 是一个用于初始化 Vec 的方法，它接受一个 Seq 作为参数，并将其转换为 Vec。VecInit 在初始化时会将 Seq 中的元素逐个赋值到 Vec 中。下面是一个示例：

```
val myVec = VecInit(Seq(1.U, 2.U, 3.U, 4.U))
```

在这个例子中，myVecInit 是一个包含 4 个元素的向量，每个元素都是 1 位宽的无符号整数。VecInit 将 Seq 中的每个元素赋值到 Vec 的对应位置。

Vec 和 VecInit 的区别

1）定义方式的区别：

Vec 通常用于动态创建和定义一个向量，其长度和类型需要在定义时指定。VecInit 用于静态初始化一个向量，其元素在初始化时已经确定。

2）使用场景

Vec 更多用于需要在运行时动态操作和修改的向量。VecInit 适用于在设计时已经确定的静态向量初始化。

在实际使用中，选择 Vec 还是 VecInit 取决于具体的需求。如果需要一个预定义的、固定的向量，使用 VecInit 更为简洁。如果需要一个可以动态操作和修改的向量，使用 Vec 更为灵活

3.5. PriorityMux

PriorityMux 在 Chisel 中用于实现优先级选择器，它根据输入信号的优先级顺序选择第一个有效的输入信号并将其传递到输出端。下面是一个 PriorityMux 的示例：

```
val selSignals = Seq(cond0, cond1, cond2, cond3)
val inputs = Seq(in0, in1, in2, in3)
val result = PriorityMux(selSignals zip inputs)
```

在这个例子中，selSignals 是一组条件信号，当 cond0 为真时，result 输出 in0；如果 cond0 为假且 cond1 为真，result 输出 in1，依此类推。如果所有条件信号都为假，PriorityMux 将输出默认值 default。

接下来的实验步骤中，我们将结合这些语法进一步探讨具体的实验案例（编码器、译码器、查找表）。

4. 实验步骤

4.1. 编码器与译码器介绍

在数字电路中，编码器和译码器是两种基本而重要的逻辑电路，它们在数字电路设计中经常出现，编码器和译码器也常被用来和其他逻辑电路结合在一起成为更复杂的逻辑电路。

编码 编码器是一种将多输入信号转换为较少输出信号的装置。它的主要功能是将多条输入信号线中的一个有意义的输入信号转换成一组唯一的二进制代码。例如，一个 4-to-2 编码器有 4 个输入和 2 个输出，如果其中一个输入为高电平，对应的输出则会生成一个特定的二进制代码，如图 1 是对应的逻辑框图，图 2 是功能框图，而表 1 则是该编码器的真值表。

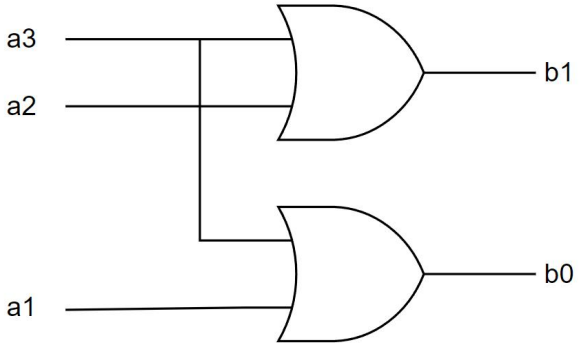


图 1 4-2 编码器的逻辑框图

根据图 1 的逻辑框图，我们可以写出输入与输出的逻辑表达式，例如 b0 =a3 || a1 ,b1 = a3 || a2 （注意，a0 信号在 4-2 编码器中实际上并不会用到，只用 a1~a3 就能满足逻辑功能），其中 || 表示 OR gate 或门，这样我们对于每一个逻辑输出都能找到一系列的逻辑输入来进行对应，并且通过逻辑表达式获得了其逻辑门上的关系，我们能够确定电路中各个逻辑门的功能。这样一来，我们就能够依据逻辑表达式中的关系，通过一系列的逻辑门来搭建电路。例如，在实际的电路设计中，我们可以使用 OR 门和 AND 门以及 NOT 门来实现这些逻辑表达式，从而构建出符合设计要求的逻辑电路。相比之下，Chisel 等硬件描述语言（HDL）则偏向于从功能和行为的角度来描述电路设计。HDL 允许设计者以更高层次的抽象来描述电路的功能，而不需要过多关注底层的逻辑门实现。例如，在 Chisel 中，我们可以使用高级的编程语法来定义模块和其内部的行为逻辑，从而生成符合设计需求的硬件电路。这种方式不仅提高了设计效率，还使得设计者能够更容易地进行电路的验证和调试。因为在 HDL 中，设计者可以通过仿真工具对电路进行功能验证，确保其符合预期的行为。这种设计方法也更加适合现代复杂电路的设计需求，因为它能够更好地管理和优化电路的复杂性。

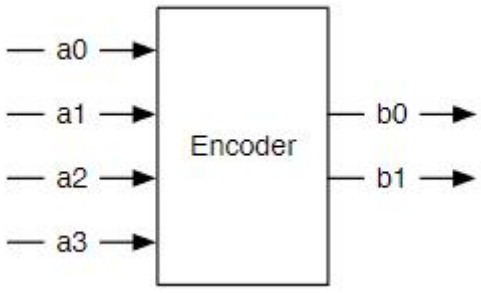


图 2 4-2 编码器功能框图

a	b
0001	00
0010	01
0100	10
1000	11
其他情况	未定义

表 1 4-2 编码器真值表

接下来我们将采用 Chisel 来实现这个编码器的功能逻辑，图 2 表明了一个 4 位独热（OneHot）输入到一个 2 位二进制输出的编码，通过观察真值表可以观察到一个编码器只在输入信号是独热的情况下正常工作，对于其它的输入，输出是未定义的。这里顺便介绍一下独热码（One-Hot Code）是一种二进制编码方式，其中每一位中只有一位是“1”，其余位都是“0”，独热码的优点是简单直观，容易实现硬件解码，且误差检测能力强。

对于上面描述的 4-2 编码器，我们可以使用下面的 Chisel 代码实现：

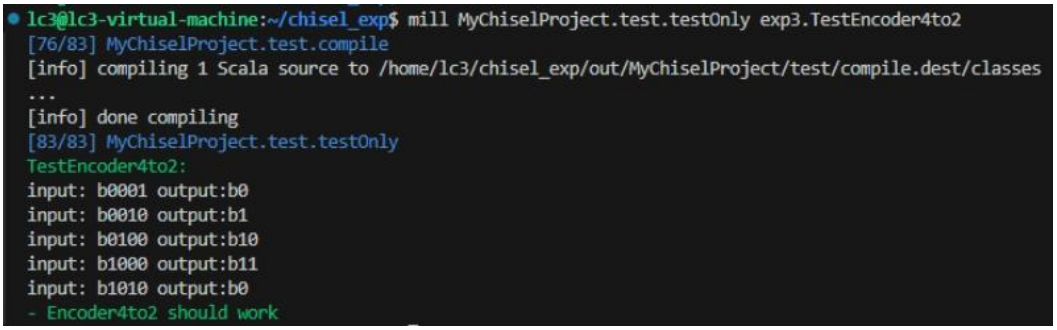
- 以下的代码中，如果 a 的输入不是独热码，那么 b 的默认输出是 b00，默认输出只会在下面的 switch 语句没有任何一项匹配到的时候才会起作用；
- Chisel 支持 switch 语法，这个与 C 语言中的 switch 类似，switch 需要接收一个进行匹配的信号，在下面的代码中，是 io.a 这个输入信号，接下来 switch 的匹配项由一个个的 is 语句组合而成，14~17 行都是要进行匹配的项，需要注意的是我们采用的是 Chisel 的 binary string literal 的方式来创建 UInt 值，在我们这一种电路中，这样编写有利于提高可读性，当然你也可以直接使用常规的方式来创建 UInt 值，例如对于"b0100".U 可以用 4.U 来代替，这是等价的。在 is 的匹配项之后，就是要满足条件后要执行的语句，这里要执行的语句就是为对应的 io.b 赋值。

```
1.  package exp3
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class Encoder4to2 extends Module {
7.    // 端口定义：
8.    val io = IO(new Bundle {
```

```
    val a = Input(UInt(4.W))
    val b = Output(UInt(2.W))
  })

  // 功能实现：
  io.b := "b00".U // Default all outputs to 0
  switch(io.a) {
    is("b0001".U) { io.b := "b00".U }
    is("b0010".U) { io.b := "b01".U }
    is("b0100".U) { io.b := "b10".U }
    is("b1000".U) { io.b := "b11".U }
  }
}
```

可以切换到实验配套项目根目录，在 src/main/scala/exp3/Encoder4to2.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestEncoder4to2
```



可以看到对于 b1010 这个输入，得到的输出是 0，这也符合我们的代码设计。

接下来介绍一下译码器，译码器则与编码器的功能相反，它将二进制代码转换为多条输出信号。一个 n-to-2^n 译码器可以将 n 位的二进制输入码转换为最多 n^2 条不同的输出信号。例如，3-to-8 译码器有 3 个输入和 8 个输出，当输入为某个特定的二进制码时，对应的输出通道会被激活。译码器在内存地址解码、数据分配以及显示驱动等领域有着广泛的应用。译码器通常采用真值表来定义输入与输出之间的映射关系，从而实现准确的译码功能。

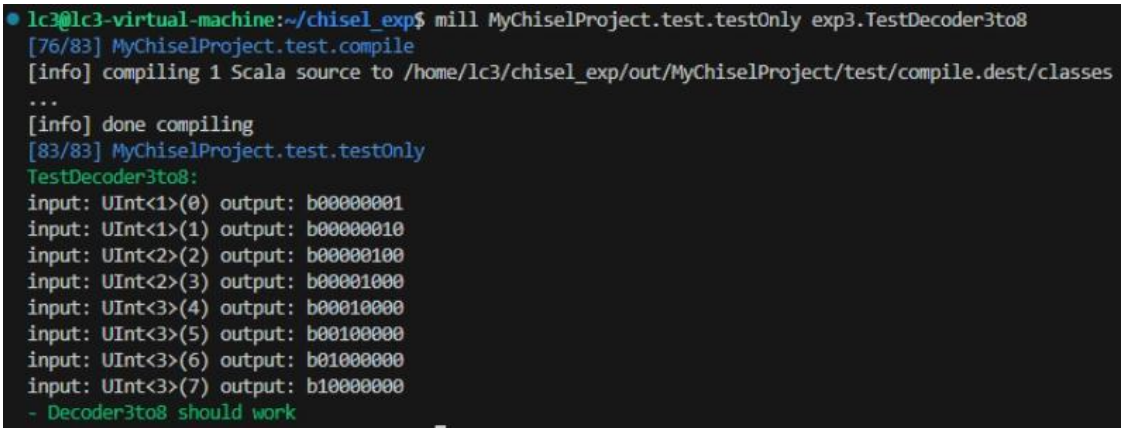
下面提供一个 3-8 译码器的 Chisel 代码的实现例子：

```
package exp3

import chisel3._
import chisel3.util._

class Decoder3to8 extends Module {
  // 端口定义：
  val io = IO(new Bundle {
    val a = Input(UInt(3.W))
```

```
    val b = Output(UInt(8.W))
  })

  // 功能实现：
  io.b := 0.U // Default all outputs to 0
  switch(io.a) {
    is("b000".U) { io.b := "b00000001".U } // 0 -> 00000001
    is("b001".U) { io.b := "b00000010".U } // 1 -> 00000010
    is("b010".U) { io.b := "b00000100".U } // 2 -> 00000100
    is("b011".U) { io.b := "b00001000".U } // 3 -> 00001000
    is("b100".U) { io.b := "b00010000".U } // 4 -> 00010000
    is("b101".U) { io.b := "b00100000".U } // 5 -> 00100000
    is("b110".U) { io.b := "b01000000".U } // 6 -> 01000000
    is("b111".U) { io.b := "b10000000".U } // 7 -> 10000000
  }
}
```

可以切换到实验配套项目根目录，在 src/main/scala/exp3/Decoder3to8.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestDecoder3to8
```



通过对比上面的 4-2 译码器的例子，我们可以看到实际上编码器与译码器就是功能相反的，在 Chisel 的功能实现上也是类似的。

除了上述使用到的 switch 语法能够实现编码器与译码器，事实上 Chisel 中还可以使用 MuxLookup 语法来实现，我们首先先看下 MuxLookup 在 Chisel 源码中的定义：

- 可以看到 MuxLookup 在调用的时候，需要指定一个用于索引的键值 key，以及默认的输出值 default；
- 还需要另外提供一个输入到输出的映射表，映射表可以使用 Scala 的 Seq 语法实现。

```
object MuxLookup extends SourceInfoDoc {

  /** @param key a key to search for
```

```
4.      * @param default a default value if nothing is found
5.      * @param mapping a sequence to search of keys and values
6.      * @return the value found or the default if not
7.      */
8.     def apply[T <: Data](key: UInt, default: T)(mapping: Seq[(UInt, T)]): T = {
9.     // ...
10.    }
11. }
```

下面我们使用 MuxLookup 来重新实现 4-2 编码器的例子：

```
1.  package exp3
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class Encoder4to2_1 extends Module {
7.      // 端口定义：
8.      val io = IO(new Bundle {
9.          val a = Input(UInt(4.W))
10.         val b = Output(UInt(2.W))
11.     })
12.
13.     // 功能实现：
14.     io.b := MuxLookup(io.a, "b00".U)(
15.         Seq(
16.             "b0001".U -> "b00".U,
17.             "b0010".U -> "b01".U,
18.             "b0100".U -> "b10".U,
19.             "b1000".U -> "b11".U
20.         )
21.     )
22. }
```

可以切换到实验配套项目根目录，在 src/main/scala/exp3/Encoder4to2_1.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestEncoder4to2_1
```

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp3.TestEncoder4to2_1
[83/83] MyChiselProject.test.testOnly
TestEncoder4to2_1:
input: b0001 output:b0
input: b0010 output:b1
input: b0100 output:b10
input: b1000 output:b11
input: b1010 output:b0
- Encoder4to2_1 should work
```

Scala 中，-> 符号可以用于组成元组，也就是上面定义的时候需要的 (UInt, T) 类型，其中的 T 是一个泛型，再使用 Seq 包裹起来就得到了我们需要的 Seq[(UInt, T)] 类型。

使用 MuxLookup 提供了更具可读性的逻辑功能表达方式，在我们的后续开发中可以多使用 Chisel 提供的这些硬件原语用于提高代码的可读性。

4.2. 优先级编码器

优先级编码器（Priority Encoder）也是一种常见的组合逻辑电路，用于从多个输入信号中选择具有最高优先级的一个，并输出其对应的二进制码。当多个输入信号同时为高电平时，优先级译码器根据预设的优先级规则选择优先级最高的信号进行编码。优先级编码器在很多场合会被用到，例如数据选择、请求仲裁等，它能够有效地简化信号处理，确保系统按优先级顺序处理任务。事实上图 1 中的逻辑框图就是一个 4-2 优先级编码器的逻辑框图。

下面是一个优先级编码器的 Chisel 代码例子，可以看到这里采用了多个 when 语句来实现，其中 in(3)具有最高的优先级，在优先级编码器中，如果我们的 4-2 编码器的输入是 b0011，正常按照 3.1 中的 4-2 编码器的 Chisel 实现，此时的输出应该是 b00，因为没有匹配任何的项，此时为输出默认值，但是优先级编码器此时的输出为 b01，因为优先级编码器只会选择具有最高优先级的来进行编码。我们可以一步步看：

- 检查输入的第 3 位 (io.in(3))，此时为 0，不满足条件。
- 检查输入的第 2 位 (io.in(2))，此时为 0，不满足条件。
- 检查输入的第 1 位 (io.in(1))，此时为 1，满足条件，因此输出 io.out 设为 1.U。
- 尽管第 0 位也为 1，但由于第 1 位有更高的优先级，输出已经在第 3 条 when 语句中确定为 1.U，不会再改变。

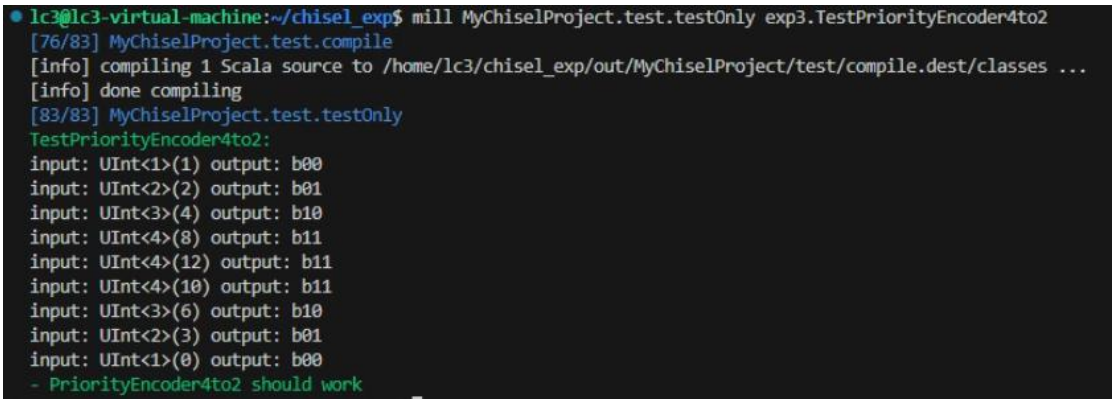
```
1.  package exp3
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class PriorityEncoder4to2 extends Module {
7.    // 端口定义：
8.    val io = IO(new Bundle {
9.      val in  = Input(UInt(4.W))
10.     val out = Output(UInt(2.W))
11.   })
12.
13.   // 功能实现：
14.   io.out := 0.U // Default output
15.
```

```
16.     when(io.in(3)) {
17.       io.out := 3.U // Highest priority
18.     }.elsewhen(io.in(2)) {
19.       io.out := 2.U
20.     }.elsewhen(io.in(1)) {
21.       io.out := 1.U
22.     }.elsewhen(io.in(0)) {
23.       io.out := 0.U // Lowest priority
24.     }
25. }
```

观察上面的 Chisel 代码我们可以看到 when 语句是支持多个 elsewhen 级联的，只需要在每个花括号之后继续使用.elsewhen 即可进行级联。

可以切换到实验配套项目根目录，在 src/main/scala/exp3/PriorityEncoder4to2.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestPriorityEncoder4to2
```



在 Chisel 中，我们还可以使用 Chisel 的 PriorityMux 来实现上述的优先级编码器的功能，PriorityMux 也是 Chisel 提供的一个硬件原语，具体代码如下：

- 可以看到 PriorityMux 的输入也是一个类似 3.1 中 MuxLookup 的映射表；
- 下面的代码中我们将映射表放在了一个 conditions 不可变变量中，这在 Chisel 中经常会看到类似的写法，能够进一步提高代码的的可读性。

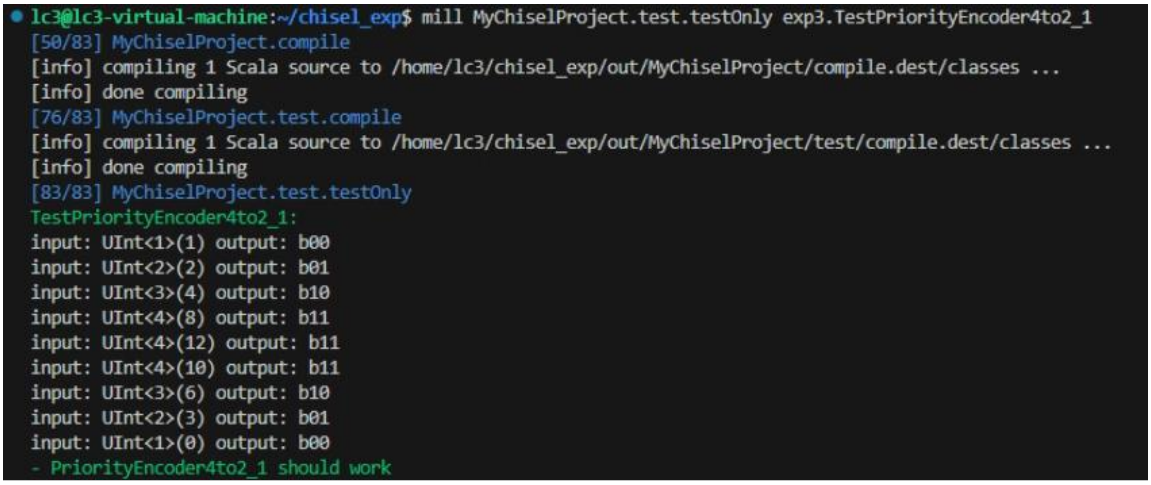
```
1.  package exp3
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class PriorityEncoder4to2_1 extends Module {
7.    // 端口定义：
8.    val io = IO(new Bundle {
9.      val in  = Input(UInt(4.W))
10.     val out = Output(UInt(2.W))
```

```
11.     })
12.
13.     // 功能实现：
14.     // 创建条件和值对的序列用于 PriorityMux
15.     val conditions = Seq(
16.         io.in(3) -> 3.U, // 最高优先级
17.         io.in(2) -> 2.U,
18.         io.in(1) -> 1.U,
19.         io.in(0) -> 0.U  // 最低优先级
20.     )
21.
22.     // 使用 PriorityMux 根据条件选择输出
23.     io.out := PriorityMux(conditions)
24. }
```

需要注意的是 PriorityMux 属于 chisel3.utils 这个 package 下，因此我们需要 import chisel3.utils._ 才能确保正常使用 PriorityMux。

可以切换到实验配套项目根目录，在 src/main/scala/exp3/PriorityEncoder4to2_1.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestPriorityEncoder4to2_1
```



4.3. 查找表

在硬件设计中，查找表（Lookup Table，LUT）是一种常用的技术，用于快速查找预先计算好的结果，类似一个 C 语言中具有初始值的数组（或者说是一个 const 的数组），我们看成是一个只读的内存（内部的内容是不可变的），这对于硬件电路来说是一个很有用的硬件逻辑组件，例如，如果我们想在硬件中生成正弦波信号，可以预先计算好一个周期内的正弦波样本值，将这些值存储在查找表中，然后通过查找表快速获取正弦波的输出值，这样可以大大简化复杂的三角函数计算。

查找表的实现原理类似编码器与译码器，都是通过输入来获得输出，因此我们也可以使用 3.1.中介绍的 switch、 MuxLookup 语法来实现，例如下面的这一个代码例子中（使用 MuxLookup 实现），实现了一个简单的 LUT，类似一个 b = a ^ a（b 是 a 的 a 次方） 的函数：

```
1.  package exp3
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class LUT extends Module {
7.    // 端口定义：
8.    val io = IO(new Bundle {
9.      val a = Input(UInt(3.W))
10.     val b = Output(UInt(8.W))
11.   })
12.
13.   // 功能实现：
14.   io.b := MuxLookup(io.a, 0.U)(
15.     Seq(
16.       0.U -> 0.U,
17.       1.U -> 1.U,
18.       2.U -> 4.U,
19.       3.U -> 9.U
20.     )
21.   )
22. }
```

可以切换到实验配套项目根目录，在 src/main/scala/exp3/LUT.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestLUT
```

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp3.TestLUT
[76/83] MyChiselProject.test.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestLUT:
Input: 0, Output: 0, Expected: 0
Input: 1, Output: 1, Expected: 1
Input: 2, Output: 4, Expected: 4
Input: 3, Output: 9, Expected: 9
Input: 4, Output: 0, Expected: 0
Input: 5, Output: 0, Expected: 0
Input: 6, Output: 0, Expected: 0
Input: 7, Output: 0, Expected: 0
- LUT should work
```

上面是一个简单的 LUT，我们可以发现这个我们的输入是连续的整数值，例如 0、1、2、3 等等，对于这种连续的索引，我们能够使用 Chisel 中的 Vec 数据结构来实现上述的 LUT 模块，Vec 是 Chisel 中用于表示固定长度的向量（数组）的数据结构，它包含相同类型的元素，并且可以通

过索引访问这些元素，例如：

```
package exp3

import chisel3._
import chisel3.util._

class LUT_1 extends Module {
  // 端口定义：
  val io = IO(new Bundle {
    val a = Input(UInt(3.W))  // 3-bit input (can represent 0 to 7)
    val b = Output(UInt(8.W)) // 8-bit output
  })

  // 功能实现：
  // Use VecInit to create a lookup table with predefined values
  val lut = VecInit(
    Seq(
      0.U(8.W),  // Value for input 0
      1.U(8.W),  // Value for input 1
      4.U(8.W),  // Value for input 2
      9.U(8.W),  // Value for input 3
      16.U(8.W), // Value for input 4
      25.U(8.W), // Value for input 5
      36.U(8.W), // Value for input 6
      49.U(8.W)  // Value for input 7
    )
  )

  // Assign the output based on the input index
  io.b := lut(io.a)
}
```

可以切换到实验配套项目根目录，在 src/main/scala/exp3/LUT_1.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp3.TestLUT_1
```

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp3.TestLUT_1
[50/83] MyChiselProject.compile
[info] compiling 2 Scala sources to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[76/83] MyChiselProject.test.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestLUT_1:
Input: 0, Output: 0, Expected: 0
Input: 1, Output: 1, Expected: 1
Input: 2, Output: 4, Expected: 4
Input: 3, Output: 9, Expected: 9
Input: 4, Output: 16, Expected: 16
Input: 5, Output: 25, Expected: 25
Input: 6, Output: 36, Expected: 36
Input: 7, Output: 49, Expected: 49
- LUT_1 should work
```

我们还可以使用一些 Scala 的语法来帮助我们写出更加简洁的代码，例如使用 map 函数，例如下面的 lut 实现了和上面代码中 lut 一样的功能，我们对下面的代码进行进一步的解释：

- `0 until 8`：是 Scala 的一个范围表达式，这段代码表示从 0 到 7 的整数序列，不包括 8；
- `.map(i => (i * i).U(8.W))`： map 是一个高阶函数，用于将一个函数应用到序列中的每个元素，并生成一个新的序列。在这个例子中，i => (i * i) 是一个匿名函数，它接收一个整数 i 并返回 i 的平方。.U(8.W) 是 Chisel 的一个方法，用于将整数转换为 8 位的无符号整数（UInt）。

```
val lutValues = (0 until 8).map(i => (i * i).U(8.W))
val lut = VecInit(luaValues)
```

上面的 luaValues 展开后就是：

```
Seq(
    0.U(8.W),
    1.U(8.W),
    4.U(8.W),
    9.U(8.W),
    16.U(8.W),
    25.U(8.W),
    36.U(8.W),
    49.U(8.W)
)
```

任务一：参考 3.1 中表 1 的内容，写一个 4-16 译码器的真值表，并且需要使用 Chisel 实现这一模块，模块名为 Decoder4to16

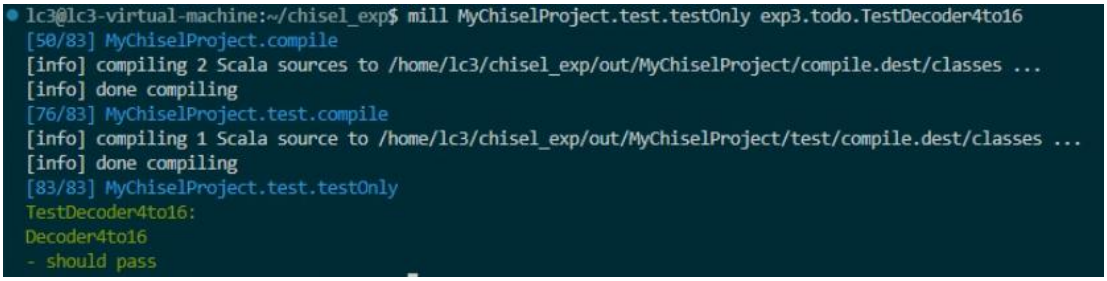
实验框架代码位于 src/main/scala/exp3/todo/Decoder4to16.scala 中（注意端口代码不可修改！）：

```
1.  package exp3.todo
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class Decoder4to16 extends Module {
7.      val io = IO(new Bundle {
8.          val in  = Input(UInt(4.W))
9.          val out = Output(UInt(16.W))
10.     })
11.
12.     // TODO: fill your code...
13. }
```

如果你完成了代码编写，请执行:

```
mill MyChiselProject.test.testOnly exp3.todo.TestDecoder4to16
```

看到下面的内容就说明测试成功：



任务二：实现一个正弦函数查找表

具体要求：

- 需要使用到上面介绍的 map 函数
- 正弦函数的离散采样点为 50 个
- 正弦函数的中间值为 125 左右（可以在这个附近）
- 模块的框架如下（src/main/scala/exp3/todo/SinLUT.scala）（注意端口代码不可修改！）：

```
1.  package exp3.todo
2.
```

```
import chisel3._
import chisel3.util._
import scala.math._

class SinLUT extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(4.W))
    val out = Output(UInt(16.W))
  })

  // TODO: fill your code...
}
```

提示：

- 正弦函数：y = sin(x) 其中 x 为角度，x 的范围 [0, 2 * Pi]
- Scala 中可以 import scala.math._ 来使用 Pi 这个预定义的值，Pi = 3.1415926...
- Scala 中可以 import scala.math._ 来使用正弦函数 sin，sin 函数的返回值可以通过 toInt 来转化为 Int

```
import scala.math._

val angle = Pi / 2
val someIntValue = (sin(angle) + 1).toInt // result is 2
```

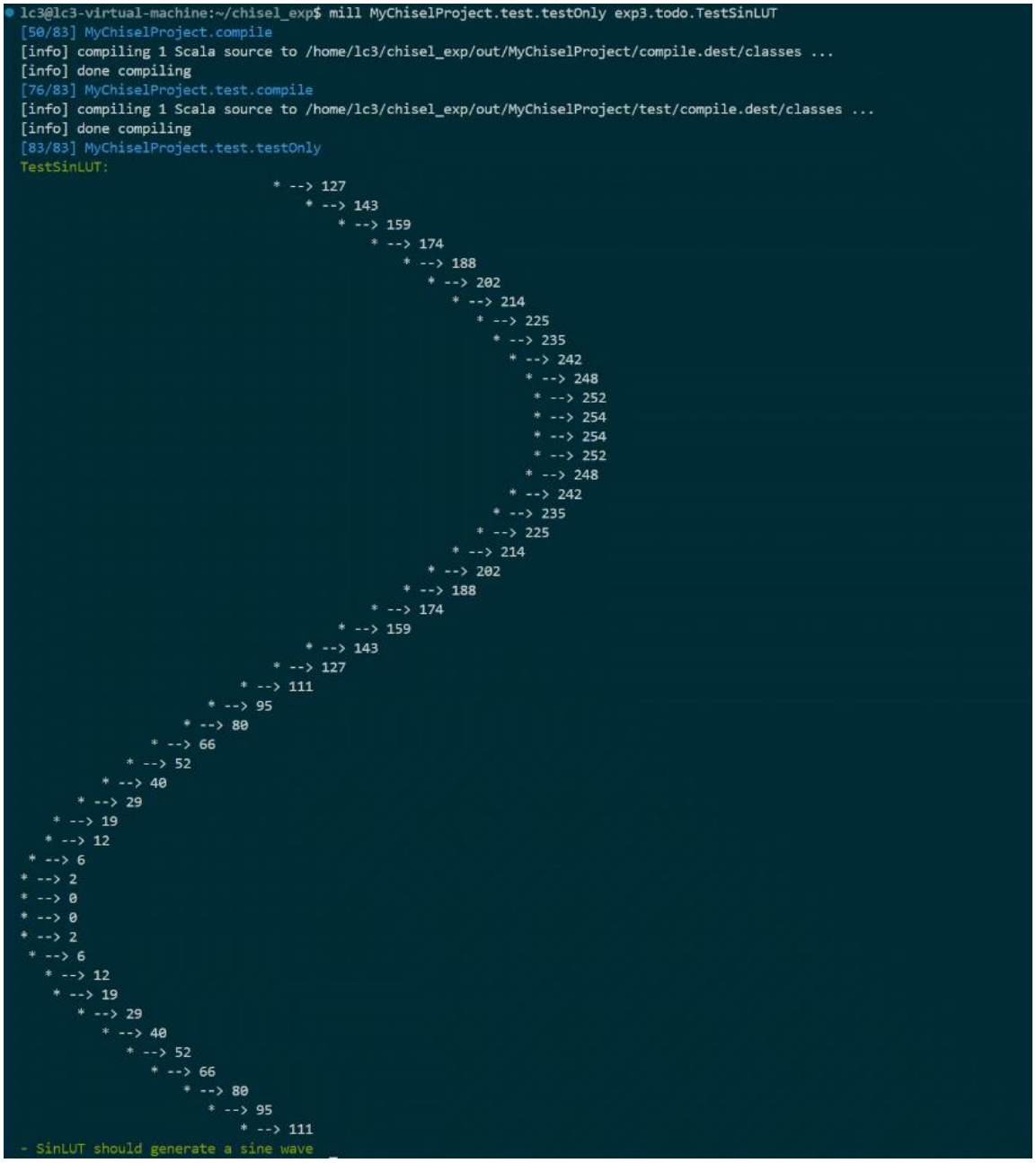
- Int 可以通过添加 .U 后缀将其转化为 Chisel 的 UInt

```
val someIntValue = 32
val chiselUIntValue = someIntValue.U
```

你可以参考下面的写法来生成一个采样点为 25，正弦函数的中间值为 100 的 lut：

```
  val lut = VecInit((0 until 25).map { i =>
    val angle = 2 * Pi * i / 25
    val sinValue = (sin(angle) * 100 + 100).toInt.U(8.W) // 将正弦值映射到 [0, 200]
    sinValue
  })
```

如果你完成了代码编写，请执行 `mill MyChiselProject.test.testOnly exp3.todo.TestSinLUT` 命令进行测试，如果测试成功你将看到下面的输出，只要得到类似正弦波的图形就算成功，不做其他要求。


lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp3.todo.TestSinLUT
[50/83] MyChiselProject.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[76/83] MyChiselProject.test.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestSinLUT:
* --> 127
* --> 143
* --> 159
* --> 174
* --> 188
* --> 202
* --> 214
* --> 225
* --> 235
* --> 242
* --> 248
* --> 252
* --> 254
* --> 254
* --> 252
* --> 248
* --> 242
* --> 235
* --> 225
* --> 214
* --> 202
* --> 188
* --> 174
* --> 159
* --> 143
* --> 127
* --> 111
* --> 95
* --> 80
* --> 66
* --> 52
* --> 40
* --> 29
* --> 19
* --> 12
* --> 6
* --> 2
* --> 0
* --> 0
* --> 2
* --> 6
* --> 12
* --> 19
* --> 29
* --> 40
* --> 52
* --> 66
* --> 80
* --> 95
* --> 111
- SinLUT should generate a sine wave