实验五：有限状态机

1. 实验目的

1.1. 学习硬件电路中状态机的设计方法；

1.2. 学习状态机的基本设计流程。

2. 实验内容

2.1. 根据状态机转移图写出状态转换表；

2.2. 学习使用 chisel 实现一个简单的 Moore 状态机和 Mealy 状态机。

3. 实验相关 Chisel 语法介绍

3.1 状态机及 Enum 语法

状态机也是常用电路，但是 Chisel 没有直接构建状态机的原语。不过，util 包中定义了一个 Enum 特质及其伴生对象。伴生对象中的 apply 方法定义如下:

```
def apply(n: Int): List[UInt]
```

它会根据参数 n 返回对应元素数的 List[UInt]，每个元素都是不同的，所以可以作为枚举值来使用。最好把枚举状态的变量名也组成一个列表，再用列表的模式匹配来进行赋值。有了枚举值后，可以通过“switch…is…is”语句来使用。其中，switch 中是相应的状态寄存器，而每个 is 分支的后面则是枚举值及相应的定义。

例如在官方代码中检测持续时间超过两个时钟周期的高电平：

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

class DetectTwoOnes extends Module {
  val io = IO(new Bundle {
    val in = Input(Bool())
    val out = Output(Bool())
  })
  val sNone :: sOne1 :: sTwo1s :: Nil = Enum(3)
  val state = RegInit(sNone)
```

```
11.
12.  io.out := (state === sTwo1s)
13.
14.  switch (state) {
15.    is (sNone) {
16.      when (io.in) {
17.        state := sOne1
18.      }
19.    }
20.    is (sOne1) {
21.      when (io.in) {
22.        state := sTwo1s
23.      } .otherwise {
24.        state := sNone
25.      }
26.    }
27.    is (sTwo1s) {
28.      when (!io.in) {
29.        state := sNone
30.      }
31.    }
32.  }
33. }
```

重点注意这句代码：

```
1.    val sNone :: sOne1 :: sTwo1s :: Nil = Enum(3)
```

这里类似于 C 语言中的枚举类型，编译器会自动按顺序从左到右把变量编码为 0.U,1.U,2.U.....

4. 实验步骤

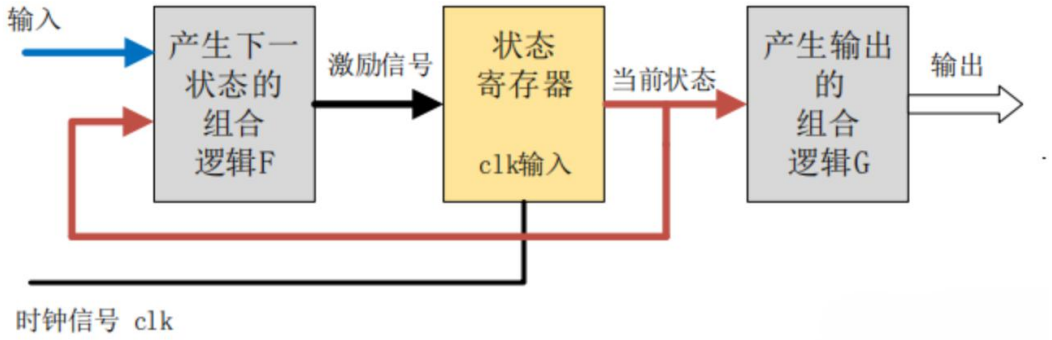
4.1 状态机介绍

有限状态机（Finite-State Machine，FSM），简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。状态机不仅是一种电路的描述工具，而且也是一种思想方法，在电路设计的系统级和 RTL 级有着广泛的应用。

Verilog 中状态机主要用于同步时序逻辑的设计，能够在有限个状态之间按一定要求和规律切换时序电路的状态。状态的切换方向不但取决于各个输入值，还取决于当前所在状态。 状态机可分为 2 类：Moore 状态机和 Mealy 状态机。

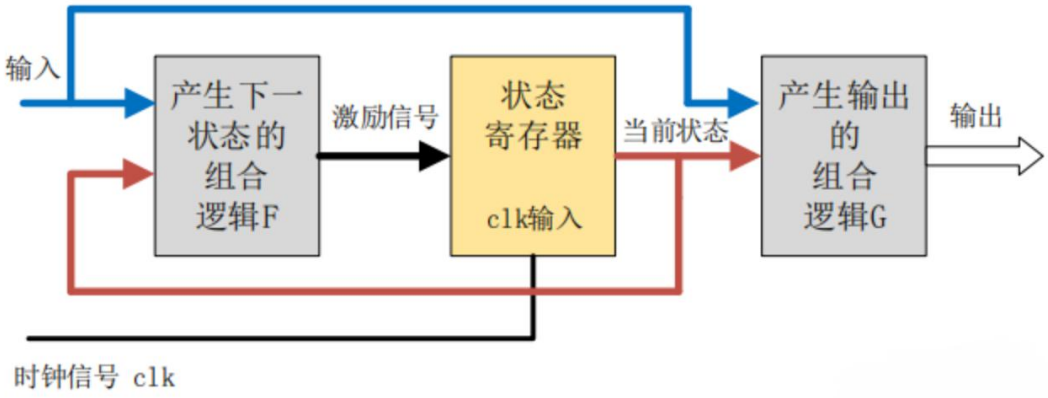
Moore 型状态机的输出只与当前状态有关，与当前输入无关。

输出会在一个完整的时钟周期内保持稳定，即使此时输入信号有变化，输出也不会变化。输入对输出的影响要到下一个时钟周期才能反映出来。这也是 Moore 型状态机的一个重要特点：输入与输出是隔离开来的。



Mealy 型状态机的输出，不仅与当前状态有关，还取决于当前的输入信号。

Mealy 型状态机的输出是在输入信号变化以后立刻发生变化，且输入变化可能出现在任何状态的时钟周期内。因此，同种逻辑下，Mealy 型状态机输出对输入的响应会比 Moore 型状态机早一个时钟周期。

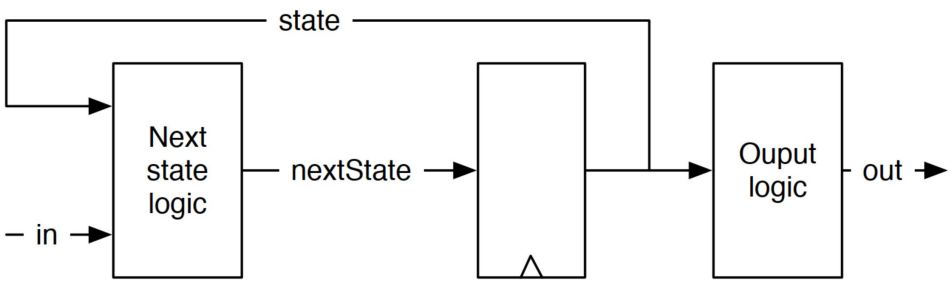


总的来说：Moore 型状态机：下一状态只由当前状态决定，即次态=f(现状，输入)，输出=f（现状）；Mealy 型状态机：下一状态不但与当前状态有关，还与当前输入值有关，即次态=f(现状，输入)，输出=f（现状，输入）；

一个 FSM 的实现主要有三个部分：

1. 保存当前状态的寄存器；
2. 基于当前状态和输入信号计算下一个状态的组合逻辑电路；
3. 计算 FSM 输出的组合逻辑。

我们先来看一个 Moore 型状态机，总体的输入和输出如图所示：

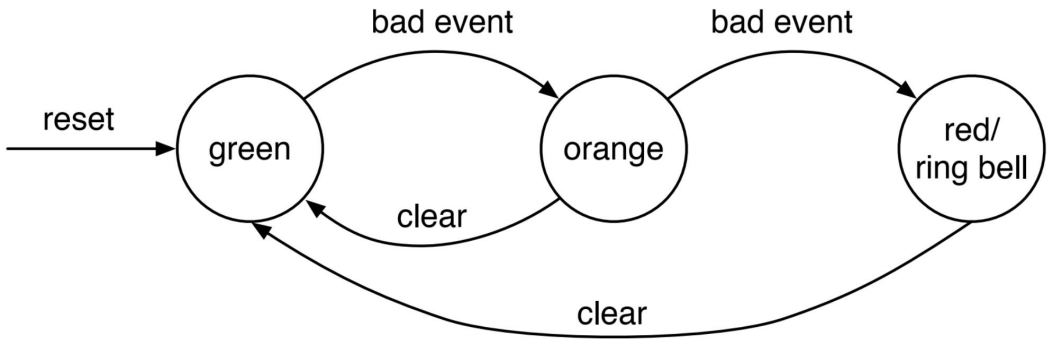


如图所示，寄存器包含了当前的状态 state，下个状态计算逻辑 Next state logic 或根据当前 state 和输入 in 来计算下一个状态 nextState，在下一个周期，nextState 就变成了 state。输出逻辑 Output logic 会计算输出 out，由于这里输出仅依赖于当前状态，则该状态机叫做 Moore 机。

4.2 状态机描述

我们接下来通过实现一个事件报警器来更加深入地学习状态机。有这样一个事件报警器，具有绿灯、橙灯和红灯三个状态和一个响铃。最开始上电时，灯为绿色，这期间可能会发生危险事件（bad event），也有可能清除警报（clear），绿灯时发生 bad event，灯将会变为橙色；橙灯时发生 bad event 时，灯将会变成红色，并且响铃（ring bell）发声。在橙灯或者红灯状态下如果发生了 clear 事件，灯将会重新变为绿色。响铃只会在红灯状态下才会工作。

以上是我们用自然语言描述这一行为，为了更加严谨，我们一般会画出状态转换图。状态转换图（State Diagram）可视化地描述 FSM 的行为，在状态转换图中，每个状态都描绘为以状态名为标签的圆圈，而状态转换用状态之间的箭头来表示。执行某个转换的条件会在状态转换的箭头以标签的形式给出。下图就是事件报警器的状态转换图：



这个 FSM 有三个状态：green、orange 和 red，表示不同等级的警告。FSM 从 green 状态开始，如果一个 bad event 来了，警告等级就会转换为 orange，如果再发生了第二个 bad event，那就进一步转换为 red，此时我们希望可以响铃，ring bell 是 FSM 唯一的输出，我们把输出加到 red 状态上，如果接受到了 clear 信号，那警告等级就会重置为 green。

我们抽象出这个状态机的输入、输出和状态：

1. 输入为 bad event 和 clear，
2. 输出为 ring bell。
3. 状态有 green、orange 和 red 三种灯的颜色。

尽管状态转换图可以可视化地描述 FSM 的功能，看起来很舒服，也很容易掌握，但状态转换表用于写代码的时候写起来更快。下表就上面警告 FSM 的状态转换表：

	Input			
State	Bad event	Clear	Next state	Ring bell
green	0	0	green	0
green	1	-	orange	0
orange	0	0	orange	0
orange	1	-	red	0
orange	0	1	green	0
red	0	0	red	1
red	0	1	green	1

表中列举了不同当前状态和输入值对应的下一个状态以及当前状态对应的输出。原则上讲，我们需要把所有状态的所有可能性都列举出来，那这个表的话就会有 $3 \times 4 = 12$ 行。但我们可以简化一些，比如如果 bad event 发生的时候，clear 信号就忽视掉了，也就是说 bad event 比 clear 有更高的优先级。输出列是有一些重复的，如果我们的 FSM 更大或者有更多的输出，那我们可以画两个表格，一个用于下个状态逻辑，另一个用于输出逻辑。

4.3 状态机的 chisel 实现

4.3.1 导入对应的库，定义输入输出端口

```
1.  import chisel3._
2.  import chisel3.util._
3.  class SimpleFsm extends Module {
4.      val io = IO(new Bundle {
5.          val badEvent = Input(Bool())
6.          val clear = Input(Bool())
7.          val ringBell = Output(Bool())
8.      })
9.
10.     // io.ringBell := TODO
11. }
```

第 1~2 行导入两个常用的库，后面我们用到的 Enum、switch 需要这两个库的支持。第 3 行我们定义 SimpleFsm 这个类，继承于 Module，这一步与前面的实验步骤一致。

根据前面小节的分析，我们的输入信号有 badEvent 和 clear，输出信号为 ringBell，这里我们均定义为 Bool 信号，当 badEvent 或 clear 输入为高时表示 badEvent 和 clear 事件发生。当 ringBell 输出为高时表示铃响事件发生。

这样我们就定义好了 module 和输入输出端口，接下来在内部实现 ringBell 的逻辑。

4.3.3 Enum 语法介绍

我们需要定义灯的状态，这里我们需要用到 Enum 语法。Enum 是枚举类型，概念与 C 语言中的枚举类型基本一致。先看一个简单的例子：

```
1.  val green :: orange :: red :: Nil = Enum(3)
```

其基本格式是 val 变量名 1 :: 变量名 2 :: ..... :: 变量名 x :: Nil = Enum(x)

表示一共有 x 个变量，各个变量名之间以::隔开，这样 chisel 会自动把变量名 1 编码为 0.U，后续依次为 1.U，2.U ..... x.U，最后用 Enum(x) 表示这是一个枚举类型，并且输入个数作为 Enum 的参数。

上述定义也等效于：

```
1.  val green = 0.U
2.  val orange = 1.U
3.  val red = 2.U
```

只不过使用 Enum 的方法更加快速简洁。

当我们定义好状态的编码后，我们需要定义一个状态寄存器，并且指定状态寄存器的初始状态为 green：

```
1.      // 状态寄存器
2.      val stateReg = RegInit(green)
```

到此，我们的代码如下，已经完成了基本的变量定义：

```
1.  import chisel3._
2.  import chisel3.util._
3.
4.  class SimpleFsm extends Module {
5.      val io = IO(new Bundle {
6.          val badEvent = Input(Bool())
7.          val clear = Input(Bool())
8.          val ringBell = Output(Bool())
9.      })
10.
```

```
    // FSM 的三种状态
    val green :: orange :: red :: Nil = Enum(3)

    // 状态寄存器
    val stateReg = RegInit(green)

    // 输出逻辑
    //io.ringBell := TODO
}
```

4.3.4 用 switch 写好基本的框架

根据状态机的转换图，我们一共有三个状态，用 3.1 小节介绍的 switch 语法，为每个状态都编写一个代码块，后续只需要根据状态转移条件，为 stateReg 变量赋值即可控制状态机的转移，代码如下：

```
    // 状态转换逻辑
    switch (stateReg) {
        is (green) {

        }
        is (orange) {

        }
        is (red) {

        }
    }
```

4.3.5 填补状态条件和输出逻辑

根据状态机转移图，当状态为 green 时，如果发生了 badEvent，则状态会变为 orange，因此在 is(green)的代码块中添加转移条件：

```
is (green) {
        when(io.badEvent) {
        stateReg := orange
    }
}
```

同理，当状态位于 orange 时，这里可能发生 badEvent 或者 clear 事件，因此在 is(orange)代码块中添加状态转移条件：

```
is (orange) {
    when(io.badEvent) {
        stateReg := red
    } .elsewhen(io.clear) {
        stateReg := green
    }
}
```

同理当状态为 red 时，添加状态转移条件：

```
is (red) {
    when(io.clear) {
        stateReg := green
    }
}
```

最后我们需要添加输出信号 ringBell，由状态转移条件可知，只有当状态机的状态为 red 时候，ringBell 时才会输出高，因此，这里只需要判断状态机的状态是否为 red 即可。最终完整代码如下：

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

class SimpleFsm extends Module {
    val io = IO(new Bundle {
        val badEvent = Input(Bool())
        val clear = Input(Bool())
        val ringBell = Output(Bool())
    })

    // FSM 的三种状态
    val green :: orange :: red :: Nil = Enum(3)

    // 状态寄存器
    val stateReg = RegInit(green)

    // 状态转换逻辑
    switch (stateReg) {
        is (green) {
            when(io.badEvent) {
                stateReg := orange
            }
        }
        is (orange) {
            when(io.badEvent) {
                stateReg := red
            } .elsewhen(io.clear) {
```

```
            stateReg := green
        }
    }
    is (red) {
        when(io.clear) {
            stateReg := green
        }
    }
}

// 输出逻辑
io.ringBell := stateReg === red
}
```

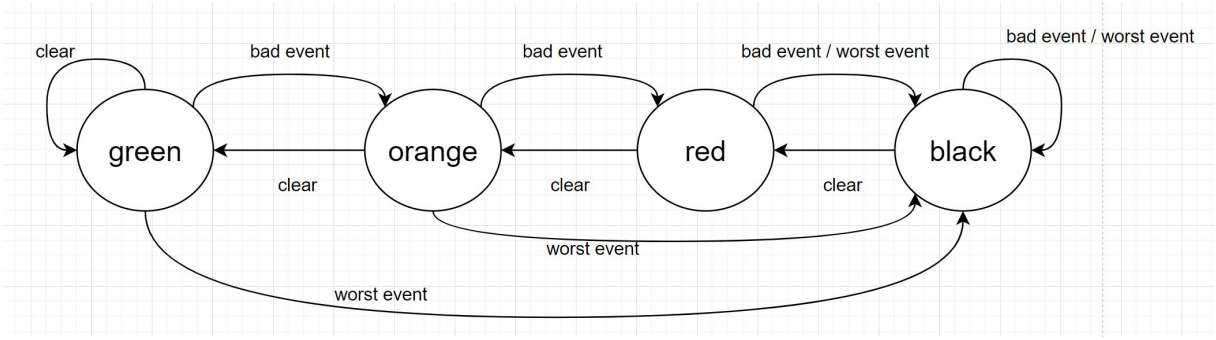
我们切换到实验配套目录的根目录，在 src/main/scala/exp5/SimpleFsm.scala 中可以看到完整的代码内容。接下来我们可以通过下面的命令来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp5.TestSimpleFsm
```

当看到以下截图时，说明代码成功运行：

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp5.TestSimpleFsm
[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
TestSimpleFsm:
- SimpleFsm should correctly switch and ring the bell
```

- 任务一：
  - 在上述的例子中，我们实现了一个 Moore 状态机，Moore 状态机的输入仅与当前的状态有关，而输入仅会改变当前的状态，接下来需要实现一个 Mealy 状态机，输出不仅与当前状态有关，更与输入有关：
  - 状态转移图如下所示：



  - 与前面不同的是，我们新增了一个状态 black，新增了一个事件 worst event，并且响铃只

有在 black 状态下发生 bad event 或者 worst event 时才会工作（这明显与上面的例子不同）。当 clear、bad event、worst event 同时发生时，优先级为：clear、worst event、bad event

- 要求：解读状态转移图，模仿 4.3 例子写出状态转换表，并通过测试。完成代码后，在工程目录下输入：

```
mill MyChiselProject.test.testOnly exp5.TestFsm
```

若看到下图显示，则表示代码已通过测试。