状态机就是 Conditionally 嵌套

例如

```
package example_module4

import chisel3._
import chisel3.util._

// 1. 定义状态枚举
object State extends ChiselEnum {
  val sIdle, sProcessing, sDone = Value
}

// 2. 定义模块 IO
class SimpleFsmIO extends Bundle {
  val start = Input(Bool()) // 输入：开始信号
  val dataIn = Input(UInt(8.W)) // 输入：一些数据（本例中未使用，仅作演示）
  val busy = Output(Bool()) // 输出：状态机是否忙碌（不在 Idle 状态）
  val dataOut = Output(UInt(8.W)) // 输出：一些数据（本例中未使用，仅作演示）
  val done = Output(Bool()) // 输出：处理完成信号
}

// 3. 定义状态机模块
class SimpleFsm extends Module {
  val io = IO(new SimpleFsmIO())

  // 导入状态枚举以便直接使用 sIdle, sProcessing, sDone
  import State._

  // 状态寄存器，复位时初始化为 sIdle
  val currentState = RegInit(sIdle)

  // 计数器，用于模拟处理时间
  val counter = RegInit(0.U(4.W)) // 假设处理需要数个周期，用4位计数器
  val PROCESSING_CYCLES = 5.U // 定义处理所需的周期数

  // --- 默认输出 ---
  // 通常为输出设置默认值，然后在特定状态下覆盖它们
  io.busy := (currentState =/= sIdle) // 只要不在Idle状态，就认为忙碌
  io.done := false.B
  io.dataOut := 0.U // 默认输出0

  // --- 下一状态逻辑（组合逻辑）---
  // 默认情况下，保持当前状态
  val nextState = WireDefault(currentState)
  // 默认情况下，计数器保持不变（除非在 sProcessing 中递增）
  // counter := counter // 这行其实不需要，因为寄存器默认保持原值

  switch(currentState) {
    is(sIdle) {
      // 在 Idle 状态：等待 start 信号
      when(io.start) {
        nextState := sProcessing
```

```
        counter := 0.U // 进入处理状态时，重置计数器
      }
    }

    is(sProcessing) {
      // 在 Processing 状态：计数器递增，模拟处理
      when(counter < PROCESSING_CYCLES) {
        counter := counter + 1.U
        nextState := sProcessing // 保持在处理状态
      }.otherwise {
        // 计数器达到目标值，处理完成
        nextState := sDone
        counter := 0.U // 可以选择在这里或进入 sDone 时重置
      }
      // 可以在这里根据 counter 或 dataIn 计算 dataOut
      // io.dataOut := io.dataIn + counter // 示例：输出可以依赖输入和内部状态
    }

    is(sDone) {
      // 在 Done 状态：发出完成信号，然后返回 Idle
      // 这个状态只持续一个周期
      nextState := sIdle
      // 在这个状态下设置完成信号
      io.done := true.B
      // 可以在这里设置最终的 dataOut 值
      // io.dataOut := some_result.U
    }
  }

  currentState := nextState

}
```

的 Circuit 为

```
Circuit( // 顶层电路定义
  SimpleFsm, // 电路名称
  List( // 电路包含的模块列表 (这里只有一个)
    Module( // 模块定义
      SimpleFsm, // 模块名称
      List( // 模块端口列表
        Port( clock,Input,ClockType),                    // 输入端口 clock，类
        Port( reset,Input,UIntType(IntWidth(1))),        // 输入端口 reset，类型
(UInt<1>)
        Port( io,Output,BundleType(Vector(               // 输出端口 io，类型为
          Field(start,Flip,UIntType(IntWidth(1))),       //   io 内部字段 'star
与 Bundle 相反 (Output Bundle 中的 Flip 是 Input)，类型 UInt<1>
          Field(dataIn,Flip,UIntType(IntWidth(8))),      //   io 内部字段 'data
(Input)，类型 UInt<8>
          Field(busy,Default,UIntType(IntWidth(1))),     //   io 内部字段 'busy
向与 Bundle 相同 (Output)，类型 UInt<1>
          Field(dataOut,Default,UIntType(IntWidth(8))),  //   io 内部字段 'data
(Output)，类型 UInt<8>
          Field(done,Default,UIntType(IntWidth(1)))      //   io 内部字段 'done
```

```
          Field(done,Default,UIntType(IntWidth(1)))      //   io 内部字段 done
(Output), 类型 UInt<1>
        )))
      ),
      Block(Vector( // 模块内部逻辑块 (一系列语句)
        // --- 寄存器定义 ---
        DefRegisterWithReset( // 定义一个带复位值的寄存器
          currentState,             // 寄存器名称
          UIntType(IntWidth(2)),    // 类型: UInt<2> (因为有 3 个状态, 需要 2 位
sProcessing=1, sDone=2)
          Reference(clock,UnknownType), // 时钟源: 连接到 clock 端口
          Reference(reset,UnknownType), // 复位信号源: 连接到 reset 端口
          UIntLiteral(0,IntWidth(1))  // 复位值: 0 (对应 sIdle)。注意字面量宽度可
被自动扩展/截断。
        ),
        DefRegisterWithReset( // 定义另一个带复位值的寄存器
          counter,                  // 寄存器名称
          UIntType(IntWidth(4)),    // 类型: UInt<4> (匹配 Chisel 代码中的 4.W)
          Reference(clock,UnknownType),
          Reference(reset,UnknownType),
          UIntLiteral(0,IntWidth(4))  // 复位值: 0
        ),

        // --- 组合逻辑和连接 (对应 Chisel 中的 := 和 Wire/Node) ---

        // io.busy := (currentState =/= sIdle)
        DefNode( // 定义一个组合逻辑节点 (类似 Verilog 的 assign 或 always_comb 内
          _io_busy_T, // 临时节点名称 (Chisel 编译器生成)
          DoPrim( // 执行一个基本操作 (Primitive Operation)
            neq, // 操作类型: 不等于 (!=)
            ArraySeq( // 操作数列表
              Reference(currentState,UnknownType), // 第一个操作数: 当前状态寄存
              UIntLiteral(0,IntWidth(1))         // 第二个操作数: 字面量 0 (sIdl
            ),
            ArraySeq(), // 参数列表 (对于 neq 为空)
            UnknownType // 结果类型 (编译器会推断, 这里是 Bool/UInt<1>)
          )
        ),
        Connect( // 连接操作 (对应 Chisel 的 :=)
          SubField(Reference(io,UnknownType),busy,UnknownType), // 连接目标: io
          Reference(_io_busy_T,UnknownType)                     // 连接源: 上面
点
        ),

        // io.done := false.B (默认值)
        Connect(
          SubField(Reference(io,UnknownType),done,UnknownType), // 连接目标: io
          UIntLiteral(0,IntWidth(1))                          // 连接源: 字面量
        ),

        // io.dataOut := 0.U (默认值)
        Connect(
          SubField(Reference(io,UnknownType),dataOut,UnknownType), // 连接目标:
          UIntLiteral(0,IntWidth(1))                               // 连接源: 字面
(0))。注意 Chisel `0.U` 默认是UInt<1>。它会被零扩展连接到 UInt<8> 的 dataOut 端口。
        ),
```

```
        ),

        // val nextState = WireDefault(currentState)
        DefWire( // 定义一个 Wire (组合逻辑信号)
          nextState,              // Wire 名称
          UIntType(IntWidth(2)) // 类型: UInt<2> (与 currentState 相同)
        ),
        Connect( // 实现 WireDefault 的默认连接
          Reference(nextState,UnknownType), // 连接目标: nextState Wire
          Reference(currentState,UnknownType) // 连接源: 当前状态寄存器的值
        ),

        // --- 状态转移逻辑 (对应 Chisel 的 switch/is) ---
        // FIRRTL 使用嵌套的 Conditionally 语句来实现 switch/when
        // if (currentState == sIdle)
        DefNode(
_T,DoPrim(asUInt,ArraySeq(UIntLiteral(0,IntWidth(1))),ArraySeq(),UnknownType))
(0) (sIdle)
        DefNode(
_T_1,DoPrim(asUInt,ArraySeq(Reference(currentState,UnknownType)),ArraySeq(),Un
_T_1 = currentState (as UInt)
        DefNode( _T_2,DoPrim(eq,ArraySeq(Reference(_T,UnknownType),
Reference(_T_1,UnknownType)),ArraySeq(),UnknownType)), // _T_2 = (_T == _T_1)
sIdle?)
        Conditionally( // 条件执行语句 (类似 if)
          Reference(_T_2,UnknownType), // 条件: currentState == sIdle
          Block(Vector( // if (currentState == sIdle) 的 True 分支
            // when (io.start)
            Conditionally( // 嵌套的条件语句
              SubField(Reference(io,UnknownType),start,UnknownType), // 条件: 
              Block(Vector( // if (io.start) 的 True 分支
                // nextState := sProcessing
                Connect( Reference(nextState,UnknownType),UIntLiteral(1,IntWid
nextState 连接到 1 (sProcessing 的编码)
                // counter := 0.U
                Connect( Reference(counter,UnknownType),UIntLiteral(0,IntWidth
counter 寄存器的 *输入* 连接到 0 (下一个周期生效)
              )),
              EmptyStmt // if (io.start) 的 False 分支 (什么也不做)
            )
          )),
          Block(Vector( // if (currentState == sIdle) 的 False 分支 (else)
            // else if (currentState == sProcessing)
            DefNode(
_T_3,DoPrim(asUInt,ArraySeq(UIntLiteral(1,IntWidth(1))),ArraySeq(),UnknownType
UInt<1>(1) (sProcessing)
            DefNode(
_T_4,DoPrim(asUInt,ArraySeq(Reference(currentState,UnknownType)),ArraySeq(),Un
_T_4 = currentState
            DefNode( _T_5,DoPrim(eq,ArraySeq(Reference(_T_3,UnknownType),
Reference(_T_4,UnknownType)),ArraySeq(),UnknownType)), // _T_5 = (currentState
            Conditionally( // 条件执行
              Reference(_T_5,UnknownType), // 条件: currentState == sProcessing
              Block(Vector( // if (currentState == sProcessing) 的 True 分支
                // when (counter < PROCESSING_CYCLES)
                DefNode( _T_6,DoPrim(lt,ArraySeq(Reference(counter,UnknownType
```

```
UIntLiteral(5,IntWidth(3))),ArraySeq(),UnknownType)), // _T_6 = (counter < 5)?
(PROCESSING_CYCLES=5, 需要3位表示)
                Conditionally( // 嵌套条件
                  Reference(_T_6,UnknownType), // 条件: counter < 5
                  Block(Vector( // if (counter < 5) 的 True 分支
                    // counter := counter + 1.U
                    DefNode( _counter_T,DoPrim(add,ArraySeq(Reference(counter,
UIntLiteral(1,IntWidth(1))),ArraySeq(),UnknownType)), // _counter_T = counter
位)
                    DefNode(
_counter_T_1,DoPrim(tail,ArraySeq(Reference(_counter_T,UnknownType)),ArraySeq(
// _counter_T_1 = tail(_counter_T, 1) (取低 4 位，丢弃可能的进位)
                    Connect(
Reference(counter,UnknownType),Reference(_counter_T_1,UnknownType)), // counte
接到加法结果
                    // nextState := sProcessing
                    Connect( Reference(nextState,UnknownType),UIntLiteral(1,In
nextState 连接到 1 (sProcessing)
                  )),
                  Block(Vector( // if (counter < 5) 的 False 分支 (else: counte
                    // nextState := sDone
                    Connect( Reference(nextState,UnknownType),UIntLiteral(2,In
nextState 连接到 2 (sDone 的编码)
                    // counter := 0.U
                    Connect( Reference(counter,UnknownType),UIntLiteral(0,IntW
counter 寄存器 *输入* 连接到 0
                  ))
                )
              )),
              Block(Vector( // if (currentState == sProcessing) 的 False 分支 (
                // else if (currentState == sDone)
                DefNode(
_T_7,DoPrim(asUInt,ArraySeq(UIntLiteral(2,IntWidth(2))),ArraySeq(),UnknownType
UInt<2>(2) (sDone)
                DefNode(
_T_8,DoPrim(asUInt,ArraySeq(Reference(currentState,UnknownType)),ArraySeq(),Un
_T_8 = currentState
                DefNode( _T_9,DoPrim(eq,ArraySeq(Reference(_T_7,UnknownType),
Reference(_T_8,UnknownType)),ArraySeq(),UnknownType)), // _T_9 = (currentState
                Conditionally( // 条件执行
                  Reference(_T_9,UnknownType), // 条件: currentState == sDone
                  Block(Vector( // if (currentState == sDone) 的 True 分支
                    // nextState := sIdle
                    Connect( Reference(nextState,UnknownType),UIntLiteral(0,In
nextState 连接到 0 (sIdle)
                    // io.done := true.B (覆盖默认值)
                    Connect(
SubField(Reference(io,UnknownType),done,UnknownType),UIntLiteral(1,IntWidth(1)
接到 1 (true)
                  )),
                  EmptyStmt // if (currentState == sDone) 的 False 分支 (理论上
要)
                )
              ))
            )
```

```
          ))
        ),

        // --- 状态寄存器更新 ---
        // currentState := nextState (在 Chisel 中隐式发生在时钟边沿)
        Connect( // 这个 Connect 定义了 currentState 寄存器在下一个时钟周期的值
          Reference(currentState,UnknownType), // 连接目标: currentState 寄存器
          Reference(nextState,UnknownType)     // 连接源: 最终计算出的 nextState
        )
      )) // End of Block
    ) // End of Module
  ) // End of Module List
) // End of Circuit
```