Experiment 7

Finite State Machines

Objective

Learn to implement complex sequential circuits involving multiple states with arbitrary state transitions in Chisel.

7.1 Finite State Machine

No sequential circuit discussion is complete without finite state machines (FSMs). Finite state machines are constructed using the following three building blocks.

- Next state logic
- State register
- Output logic

At the start, the *state register* is initialized with the default state. The *next state logic* can be implemented by using either 'switch' or 'when' blocks or both. When using both, a possible scenario is where switch case will have 'when' block nested within it. The switch cases will determine the current state, while the 'when' blocks within it will be responsible for the change of state based on the input and current state. Finally, the *output logic* will be responsible to update the output.

7.1.1 FSM Types

The output logic implementation determines the type of finite state machine. In general, FSMs can be one of the following two types.

- **Mealy state machines:** Output depends on both state and input (see Figure 7.1(a) for illustration).
- **Moore state machines:** Output depends on state only (see Figure 7.1(b) for illustration).

7.1.2 An Example FSM

Figure 7.2 shows an FSM that can be used to detect a sequence ('110' in this case) of bits in a bitstream. This state machine has been implemented in Listing 7.1. Type enumeration is used to list and define the names for different state. The 'switch' and 'when' constructs are used to implement the state transition, as discussed previously.

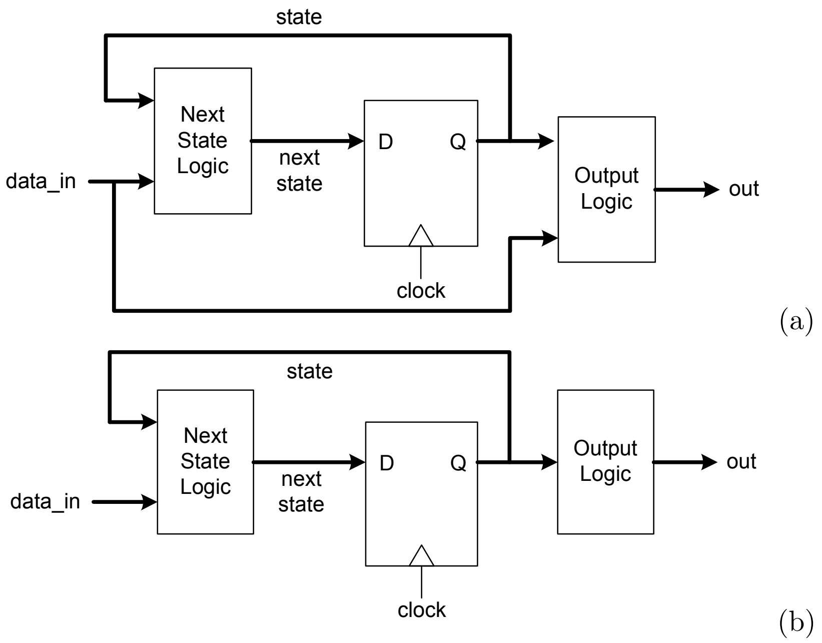

Figure 7.1: (a) Mealy type state machine, and (b) Moore type state machine.

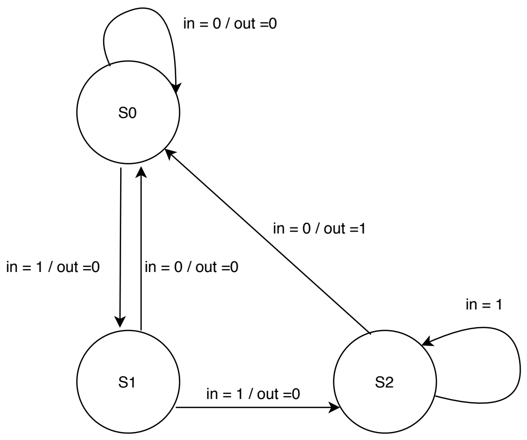

Figure 7.2: Sequence detector using an FSM.

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{
  ChiselFlatSpec, Driver, PeekPokeTester
}
import chisel3.experimental.ChiselEnum

//Sequence to detect is  110
class Detect_Seq extends Module {
  val io = IO(new Bundle {
    val in = Input(Bool())
    val out = Output(Bool())
  })

  val s0 :: s1 :: s2 :: Nil = Enum(3)   //Enumeration type
  val state = RegInit(s0)               //state = s0
```

```
    io.out := (state === s2) & (!io.in)   //Mealy type state machine

    switch (state) {
        is (s0) {
            when (io.in) {
                //move to next state when input is  1
                state := s1
            }
        }
        is (s1) {
            when (io.in) {
                //move to next state when input is  1
                state := s2
            } .otherwise {
                state := s0
            }
        }
        is (s2) {
            when (!io.in) {
                //move to default state when input is zero
                // otherwise stay here because input sequence is 111
                state := s0
            }
        }
    }
}
println(chisel3.Driver.emitVerilog(new Detect_Seq))
```

Listing 7.1: FSM for sequence detection.

7.2 Example: Up-down Counter

Next we discuss the implementation of an up-down counter using state machine. This counter is different from the one discussed previously in that it effectively generates a triangular waveform (when its count is plotted against time). The state transition diagram for the counter is shown in Figure 7.3. We observe that, once started, the counter keeps on toggling between up and down states. Listing 7.2 provides a possible implementation of the up-down counter and its test is given in Listing 7.3. The waveforms for different signals are shown in Figure 7.4.

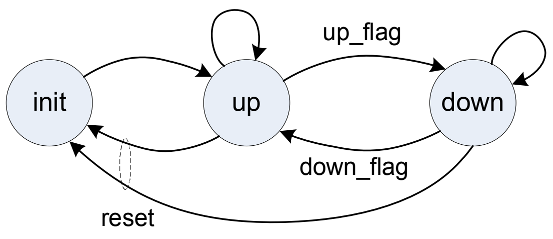

Figure 7.3: State transition diagram for up-down counter.

```
// up-down counter implementation

package LM_Chisel

import chisel3._
import chisel3.util._

class CounterUpDown(n: Int) extends Module {
    val io = IO(new Bundle {
        val data_in = Input(UInt(n.W))
        val out = Output(Bool())
    })

    val counter = RegInit(0.U(n.W))
    val max_count = RegInit(6.U(n.W))

    val init :: up :: down :: Nil = Enum(3)   // Enumeration type
    val state = RegInit(init)                 // state = init
    val up_flag = (counter === max_count)
    val down_flag = (counter === 0.U)

    switch (state) {
        is (init) {
            state := up                          // on reset
        }

        is (up) {
            when (up_flag) {
                state := down
                // start count down immediately on up_flag
                counter := counter - 1.U
            }.otherwise {
                counter := counter + 1.U
            }
        }

        is (down) {
            when (down_flag) {
                state := up
                counter := counter + 1.U
                max_count := io.data_in    // load the counter
            }.otherwise {
                counter := counter - 1.U
            }
        }
    }
    io.out := up_flag | down_flag
}
object CounterUpDown_generate extends App {
    chisel3.Driver.execute(args, () => new CounterUpDown(8))
}
```

Listing 7.2: Up-down counter implementation.

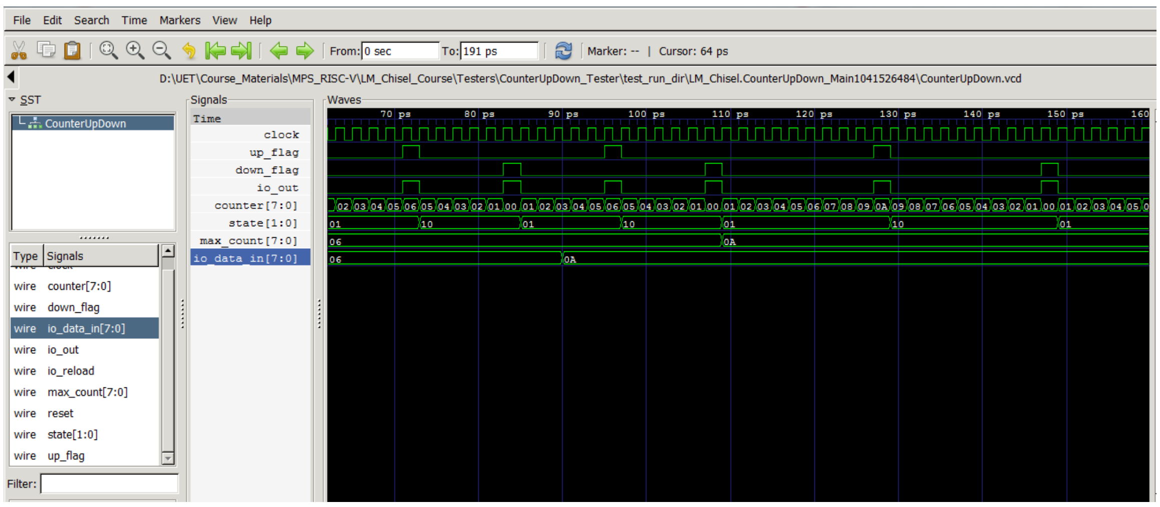

Figure 7.4: Up-down counter signal waveforms.

```
package LM_Chisel
import chisel3._
import chisel3.iotesters.{Driver, PeekPokeTester}

class TestCounterUpDown(c: CounterUpDown) extends PeekPokeTester(c) {
   var data_in = 6
   poke(c.io.data_in, data_in.U)
   step(40)
   data_in = 10
   poke(c.io.data_in, data_in.U)
   step(50)
}

// object for tester class
object CounterUpDown_Main extends App {
    iotesters.Driver.execute(Array("--is-verbose",
    "--generate-vcd-output","on", "--backend-name", "firrtl"),
    () => new CounterUpDown(8)) {c => new TestCounterUpDown(c)}
}
```

Listing 7.3: Up-down counter test.

7.3 Example: UART Transmitter

Finally we discuss the implementation of UART transmitter module, which involves, apart from state machine, many different Chisel constructs that we have discussed so far. The UART transmitter is responsible for serial data transmission at a predefined baudrate (bit rate). The baud rate frequency is derived from the clock, using the baud divisor parameter provided by the user, as part of UART transmitter implementation. Baud divisor along with other configuration parameters are passed by defining a parameter class as can be observed from the Listing 7.4. The implementation of the UART transmitter is provided in Listing 7.5.

```
import chisel3._
import chisel3 . stage . ChiselStage
import chisel3.util._

case class UART_Params(
    dataBits:    Int = 8,
    stopBits:    Int = 2,
    divisorBits: Int = 5,
    oversample:  Int = 2,
    nSamples:    Int = 3,
    nTxEntries:  Int = 4,
    nRxEntries:  Int = 4) {
  def oversampleFactor = 1 << oversample
  require(divisorBits > oversample)
  require(oversampleFactor > nSamples)
}
```

Listing 7.4: UART configuration parameters.

```

class UART_Tx(c: UART_Params) extends Module {
  val io = IO(new Bundle {
    val en     = Input(Bool())
    val in     = Flipped(Decoupled(UInt((c.dataBits).W)))
    val out    = Output(Bool())
    val div    = Input(UInt((c.divisorBits).W))
    val nstop  = Input(UInt((c.stopBits).W))
  })
  // pulses generated at baud rate using prescaler
  val prescaler = RegInit(0.U((c.divisorBits).W))
  val pulse     = (prescaler === 0.U)
  private val n = c.dataBits + 1

  val counter = RegInit(0.U((log2Floor(n + c.stopBits)+1).W))
  val shifter = Reg(UInt(n.W))
  val out     = RegInit(true.B)
  io.out      := out

  val busy    = (counter =/= 0.U)
  val state1  = io.en && !busy
  val state2  = busy
  io.in.ready := state1

  when(state1) {
    shifter  := Cat(io.in.bits, false.B)
    counter  := Mux1H(
      (0 until c.stopBits).map(i => (io.nstop === i.U) -> (n+i+2).U)
    )
  }

  when(state2) {
    prescaler := Mux(pulse, (io.div - 1.U), prescaler - 1.U)

    when(pulse) {
      counter := counter - (1.U)
      shifter := Cat(true.B, shifter >> 1)
      out := shifter(0)
    }
  }
}

// Instantiation of the UART_Tx module for Verilog generator
object UART_Tx_generate extends App {
val param = UART_Params()
  chisel3.Driver.execute(args, () => new UART_Tx(param))
}
```

Listing 7.5: UART transmitter implementation.

7.4 Arbiter

An arbiter is a device that follows producer/consumer model and is responsible to sequence n producers to one consumer. There are different types of arbiters depending on the policies they implement. The Chisel library provides two types of arbiters.

7.4.1 Priority Arbiter

The priority arbiter in Chisel utilities library is referred as *Arbiter*. When ever we make an arbiter we need to specify the number of requestors and the data type with width of the data coming from requestors. Each reuestor should be Decoupled and Flipped before connecting to the arbiter. The output of the arbiter is also decoupled. A priority arbiter supporting 2 requestors is implemented in Listing 7.6 using `Arbiter` construct.

```
val arb_priority = Module(new Arbiter(UInt(), 3))

// connect the inputs to different producers
arb_priority.io.in(0) <> producer0.io.out
arb_priority.io.in(1) <> producer1.io.out
arb_priority.io.in(2) <> producer2.io.out

// connect the output to consumer
consumer.io.in <> arb_priority.io.out
```

Listing 7.6: Priority arbiter using `Arbiter` construct.

7.4.2 Round Robin Arbiter

Round robin arbiter is an arbiter that implements round robin policy for arbitration. A round robin arbiter can be implemented using the `RRArbiter` construct as illustrated in Listing 7.7.

```
val arb_noPriority = Module(new RRArbiter(UInt(), 3))

// connect the inputs to different producers
arb_noPriority.io.in(0) <> producer0.io.out
arb_noPriority.io.in(1) <> producer1.io.out
arb_noPriority.io.in(2) <> producer2.io.out

// connect the output to consumer
consumer.io.in <> arb_noPriority.io.out
```

Listing 7.7: Round robin arbiter using `RRArbitor` construct.

7.5 Exercises

Exercise 1: Connect `Arbiter` with two `Queues` at the input and perform testing using `PeekPokeTester`.

7.6 Assignments

Task 1: The state transition diagram of an FSM is shown in the Figure 7.5. It has six states, four inputs, and one output. According to state transition Table 7.1, `f1` has high priority than `r1` and `f2` has higher priority than `r2`. The output will be `3` if the state variable reaches `S2` by keeping `f1` high for two clock cycles that may not be high for consecutive cycles but it must be insured that`r1` should not be high in between them. Same is true for `f2` and `r2` but output in this case is `7`. Write a Chisel program that implements the given FSM.

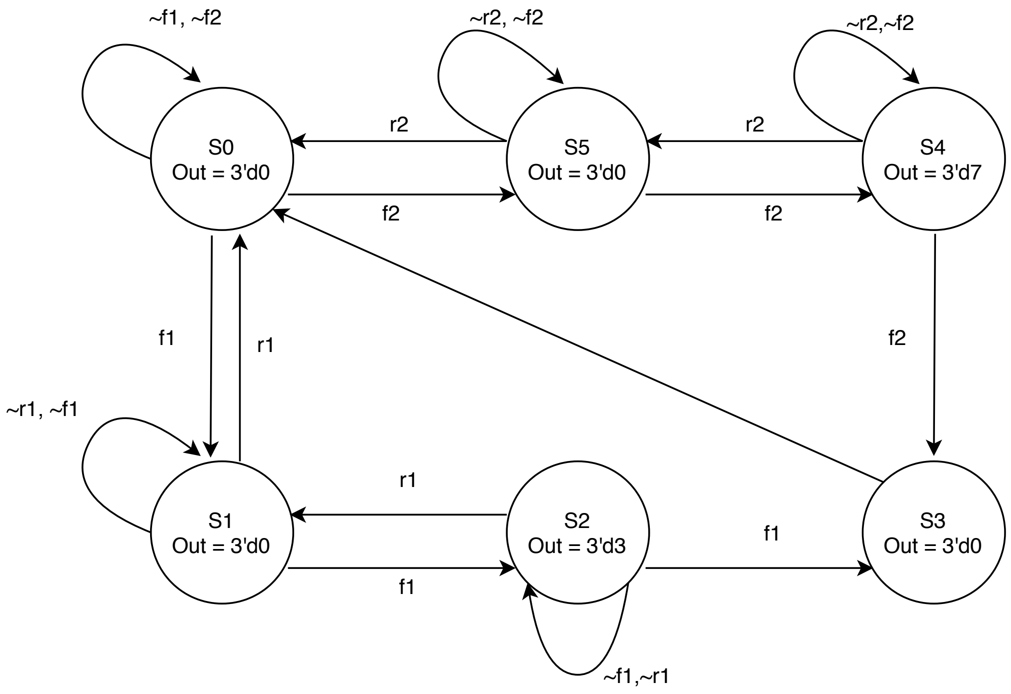

Figure 7.5: FSM forward and backward transitions.

```
package Lab7
import chisel3._
import chisel3.util._
import chisel3.iotesters.{
  ChiselFlatSpec,Driver,PeekPokeTester
}
```

Table 7.1: State Table.

Inputs				Output	State Elements	
f1	f2	r1	r2	Out	State	State next
0	0	x	x	0	S0	S0
1	0	x	x	0	S0	S1
0	1	x	x	0	S0	S5
1	1	x	x	0	S0	S1
0	x	0	x	0	S1	S1
1	x	x	x	0	S1	S2
0	x	1	x	0	S1	S0
0	x	0	x	3	S2	S2
1	x	x	x	3	S2	S3
0	x	1	x	3	S2	S1
x	x	x	x	0	S3	S0
x	1	x	x	7	S4	S3
x	0	x	0	7	S4	S4
x	0	x	1	7	S4	S5
x	1	x	x	0	S5	S4
x	0	x	0	0	S5	S5
x	0	x	1	0	S5	S0

```
class My_Queue extends Module{
   ///your code begin



   ///your code end here
}
```

Listing 7.8: Skeleton code for FSM.

Task 2: Design an FSM for decoding Manchester encoded input stream of data and place the output in the output register. Register must contains recently received eight bits of data which means you have to update it on per clock edge. Sampling must be done on the posedge of clocks. Required signals are given in the Figure 7.6 .

```
package Lab7
import chisel3._
import chisel3.util._
import chisel3.iotesters.{
   ChiselFlatSpec, Driver, PeekPokeTester
}
import chisel3.experimental.ChiselEnum
import chisel3.experimental.{
   withClock, withReset, withClockAndReset
}
import chisel3.experimental.BundleLiterals._

//optional, you can use compaion object
object Manchester_Encoding{
   //optional
```

```
    //your code
}

class Manchester_Encoding extends Module {
    import Manchester_Encoding.State
    import Manchester_Encoding.State._
    val io = IO(new Bundle {
        val in  = Input(UInt(1.W))
        val start = Input(Bool())
        val out = Output(UInt(8.W))
        val flag = Output(UInt(1.W))
    })
    //your code here
}
```

Listing 7.9: Skeleton code for Task 2.

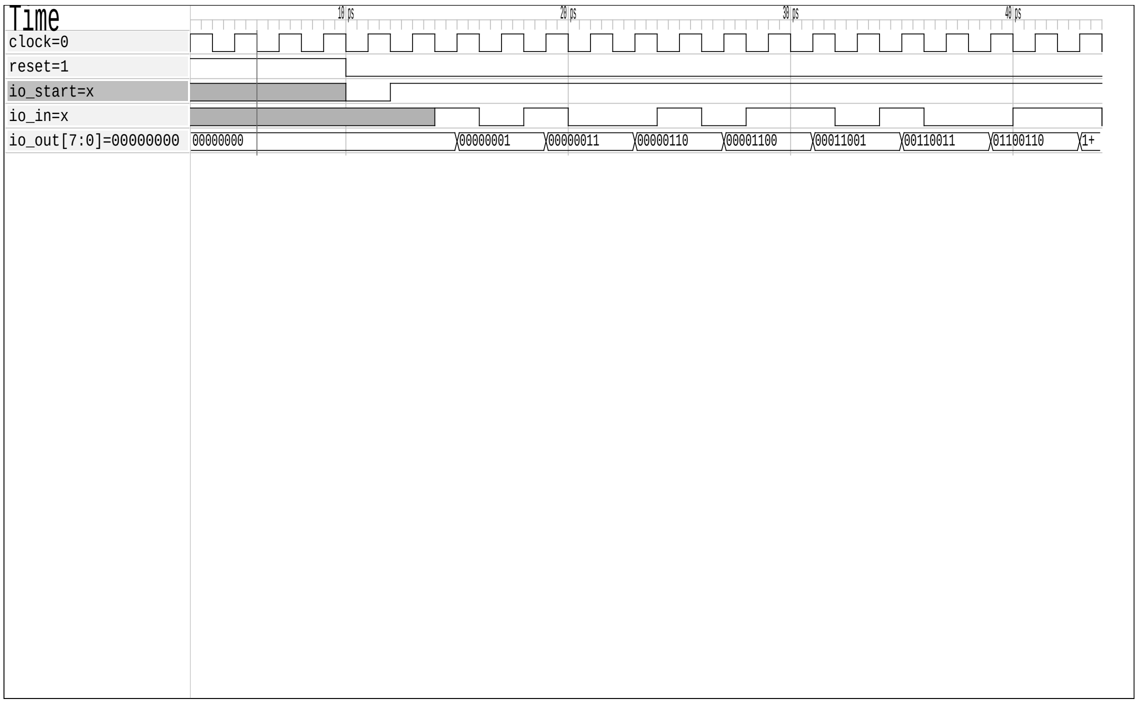

Figure 7.6: Decoupled and Queues.

Task 3: What's the difference between `Arbiter` and `RRArbiter`? Your answer should include a portion relating to the members of each class and drive your answer from hardware prespective. Suggestion: Go to the chisel-lang.org and look for API in the library under util. Also, see Verilog of both `Arbiter` and `RRArbiter`. For further details see: `https://www.chisel-lang.org/api/latest/`