Chisel: Finite State Machines

Muhammad Tahir

Lecture 7

Contents

Finite State Machines

Finite state machines (FSM) are constructed using three building blocks

- Next state logic
- State register
- Output logic

Finite State Machines

Finite state machines (FSM) are constructed using three building blocks

- Next state logic
- State register
- Output logic

State machines can be of two types

- Mealy state machines
- Moore state machines

Mealy State Machines

- Output of Mealy state machine depends on both state and input
- Can update the output faster?
- Is it synchronous or asynchronous?

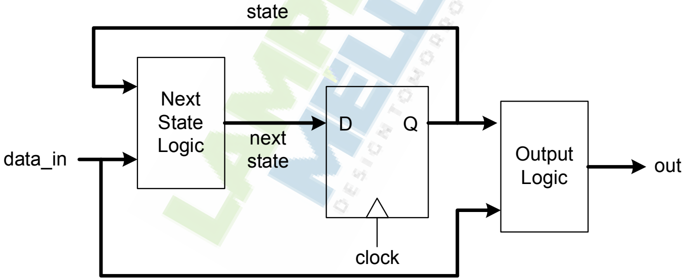

Figure: Mealy type state machine.

Moore State Machines

- Output of Moore state machine depends on state only
- Is it synchronous or asynchronous?

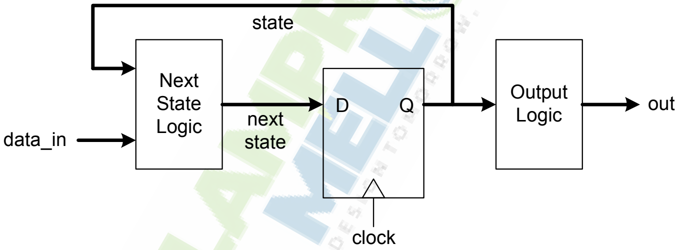

Figure: Moore type state machine.

State Machine Construct

State machines can be constructed using Enum and switch construct

```
// The set of states
val s0 :: s1 :: s2 :: s3 :: s4 :: Nil = Enum(5)

// The state register
val state = RegInit(s0)

// Next state logic
switch (state) {
      is (s0){
            // state transition logic for s0
            // corresponds to outgoing arrows from s0
      }

      is (s1) {

      }

      ...

      is (s4) {
            // state transition logic for s4
      }
}
```

Edge Detector

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

class Edge_FSM extends Module {
    val io = IO(new Bundle{
        val sin = Input(Bool())
        val edge = Output(Bool())
    })

    // Detect the edge
    io.edge = !io.sin & RegNext(io.sin)
}
```

Edge Detector Cont'd

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

class Edge_FSM extends Module {
      val io = IO(new Bundle{
            val sin = Input(Bool())
            val edge = Output(Bool())
      })

      val zero :: one :: Nil = Enum(2) // The two states
      val state = RegInit(zero)        // The state register
      io.edge := false.B               // default value for output

      // Next state and output logic
      switch (state) {

            is(zero) {
                  when(io.sin) {
                        state := one
                  }
            }

            is(one) {
                  when(!io.sin) {
                        state := zero
                        io.edge := true.B
                  }
            }
      }
}
```

Up Down Counter

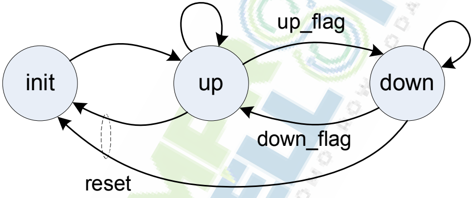

Figure: State transition diagram for up-down counter.

- What type of state machine is this?

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
```

Up Down Counter Cont'd

```
      }.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))
}
```

Up Down Counter Cont'd

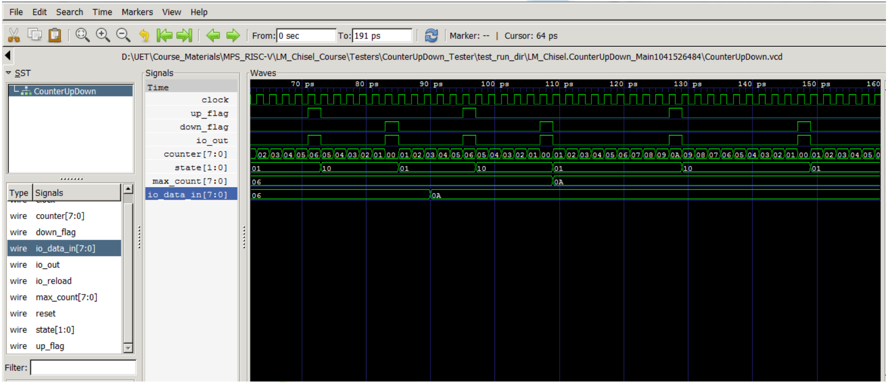

Figure: Up-down counter signal waveforms.

UART Transmit

```
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)
}
```

UART Transmit Cont'd

```
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)
```

UART Transmit Cont'd

```
  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))
}
```

Arbiter

- A hardware module following data producer/consumer model
- Sequences n producers to 1 consumer
- Connects one of the producers, at a given time, to the consumer
- Connectivity follows a certain priority mechanism

Chisel: Arbiter

```
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
```

- An arbiter module with priority
- Lower indexed producer is given higher priority
- Uses `DecoupledIO` interface for input and output connectivity

Chisel: RRArbiter

```
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
```

- An arbiter module with no or equal priority
- Each producer gets its turn in a round robin fashion

Reading List I

- Consult [chisel3, 2020] for further details
- Read Chapter 8 of [Schoeberl, 2019]

References

chisel3 (2020).
Chisel3 library reference.
https://www.chisel-lang.org/api/latest/chisel3/util.

Schoeberl, M. (2019).
Digital Design with Chisel.
Kindle Direct Publishing.