

Finite-State Machines

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

March 10, 2022

Overview

- ▶ Debugging with waveforms
- ▶ Fun with counters
- ▶ Finite-state machines
- ▶ Collection with Vec

Organization and Lab Work

- ▶ Labs are in B308-IT127, B308-IT017
- ▶ Use both rooms, you have more space in two rooms
- ▶ How did the testing lab work? Did you find both bugs?
- ▶ This week the 7-segment decoder
- ▶ A lot can be done with simulation and testing
- ▶ But, at the end of the day I want to see a vending machine in an FPGA

Testing with Chisel

- ▶ A test contains
 - ▶ a device under test (DUT) and
 - ▶ the testing logic
- ▶ Set input values with `poke`
- ▶ Advance the simulation with `step`
- ▶ Read the output values with `peek`
- ▶ Compare the values with `expect`
- ▶ Import following packages

```
import chisel3._  
import chiseltest._  
import org.scalatest.flatspec.AnyFlatSpec
```

An Example DUT

- ▶ A device-under test (DUT)
- ▶ Just 2-bit AND logic

```
class DeviceUnderTest extends Module {  
  val io = IO(new Bundle {  
    val a = Input(UInt(2.W))  
    val b = Input(UInt(2.W))  
    val out = Output(UInt(2.W))  
  })  
  
  io.out := io.a & io.b  
}
```

A ChiselTest

- ▶ Extends class `AnyFlatSpec` with `ChiselScalatestTester`
- ▶ Has the device-under test (DUT) as parameter of the `test()` function
- ▶ Test function contains the test code
- ▶ Testing code can use all features of Scala
- ▶ Is placed in `src/test/scala`
- ▶ Is run with `sbt test`

A Simple Tester

- ▶ Just using println for manual inspection

```
class SimpleTest extends AnyFlatSpec with
  ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      println("Result is: " +
        dut.io.out.peek().toString)
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      println("Result is: " +
        dut.io.out.peek().toString)
    }
  }
}
```

A Real Tester

- Poke values and expect some output

```
class SimpleTestExpect extends AnyFlatSpec
  with ChiselScalatestTester {
    "DUT" should "pass" in {
      test(new DeviceUnderTest) { dut =>
        dut.io.a.poke(0.U)
        dut.io.b.poke(1.U)
        dut.clock.step()
        dut.io.out.expect(0.U)
        dut.io.a.poke(3.U)
        dut.io.b.poke(2.U)
        dut.clock.step()
        dut.io.out.expect(2.U)
      }
    }
  }
```


Generating Waveforms

- ▶ Waveforms are timing diagrams
- ▶ Good to see many parallel signals and registers
- ▶ Or setting an attribute for the test() function

```
test(new DeviceUnderTest)  
  .withAnnotations(Seq(WriteVcdAnnotation))
```

- ▶ IO signals and registers are dumped
- ▶ Option --debug puts all wires into the dump
- ▶ Generates a .vcd file
- ▶ Viewing with GTKWave or ModelSim

A Self-Running Circuit

- ▶ Count6 is a self-running circuit
- ▶ Needs no stimuli (poke)
- ▶ Just run for a few cycles

```
test(new Count6) { dut =>  
    dut.clock.step(20)  
}
```

Call the Tester for Waveform Generation

- ▶ The complete test
- ▶ Note the `.withAnnotations(Seq(WriteVcdAnnotation))`

```
class Count6WaveSpec extends AnyFlatSpec with
  ChiselScalatestTester {
  "CountWave6 " should "pass" in {
    test(new
      Count6).withAnnotations(Seq(WriteVcdAnnotation))
      { dut =>
        dut.clock.step(20)
      }
  }
}
```

Display Waveform with GTKWave

- ▶ Run the tester: `sbt test`
- ▶ Locate the `.vcd` file in `test_run_dir/...`
- ▶ Start GTKWave
- ▶ Open the `.vcd` file with
 - ▶ File – Open New Tab
- ▶ Select the circuit
- ▶ Drag and drop the interesting signals

Counters as Building Blocks

- ▶ Counters are versatile tools
- ▶ Count events
- ▶ Generate timing ticks
- ▶ Generate a one-shot timer

Counting Up and Down

► Up:

```
val cntReg = RegInit(0.U(8.W))
```

```
cntReg := cntReg + 1.U  
when(cntReg === N) {  
    cntReg := 0.U  
}
```

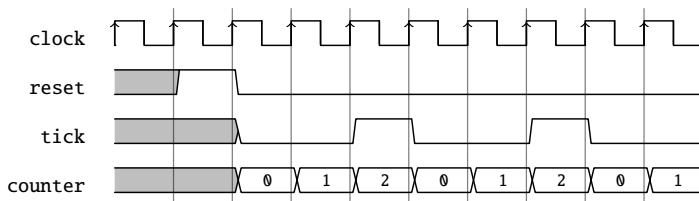
► Down:

```
val cntReg = RegInit(N)
```

```
cntReg := cntReg - 1.U  
when(cntReg === 0.U) {  
    cntReg := N  
}
```

Generating Timing with Counters

- ▶ Generate a tick at a lower frequency
- ▶ We used it in Lab 1 for the blinking LED
- ▶ Use it for driving the display multiplexing at 1 kHz



The Tick Generation

```
val tickCounterReg = RegInit(0.U(32.W))
val tick = tickCounterReg == (N-1).U

tickCounterReg := tickCounterReg + 1.U
when (tick) {
    tickCounterReg := 0.U
}
```

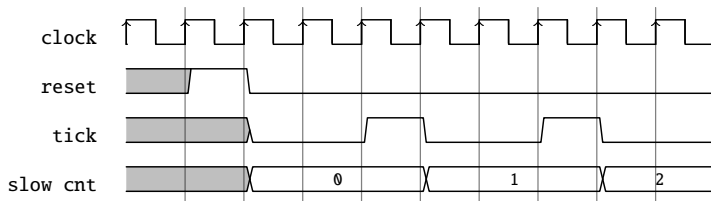

Using the Tick

- ▶ A counter running at a *slower frequency*
- ▶ By using the tick as an enable signal

```
val lowFrequCntReg = RegInit(0.U(4.W))  
when (tick) {  
    lowFrequCntReg := lowFrequCntReg + 1.U  
}
```

The *Slow* Counter

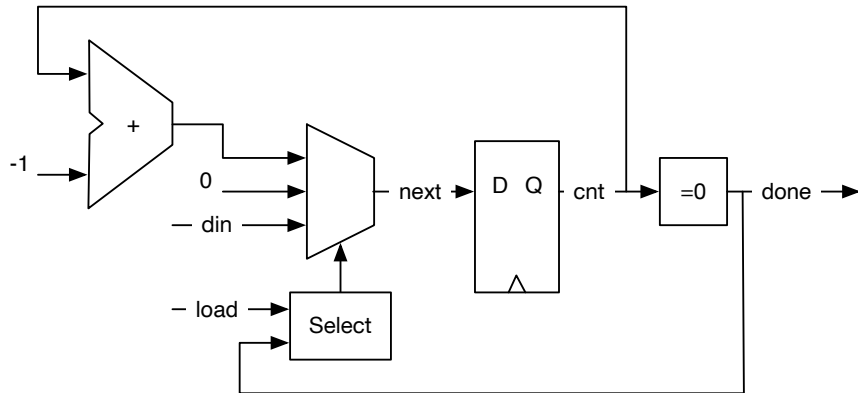
- Incremented every tick



A Timer

- ▶ Like a kitchen timer
- ▶ Start by loading a timeout value
- ▶ Count down till 0
- ▶ Assert done when finished

One-Shot Timer

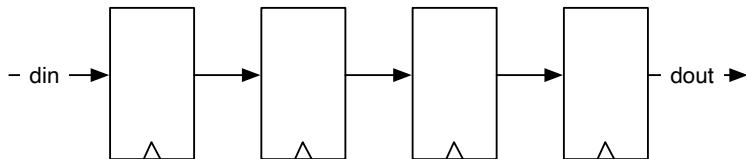


One-Shot Timer

```
val cntReg = RegInit(0.U(8.W))
val done = cntReg === 0.U

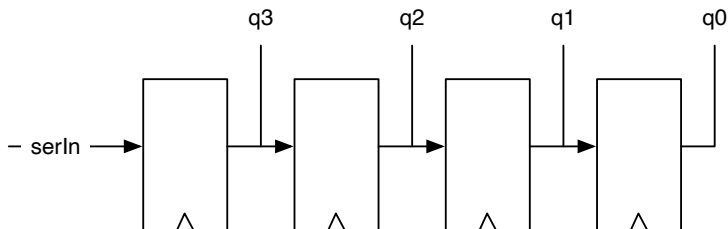
val next = WireDefault(0.U)
when (load) {
    next := din
} .elsewhen (!done) {
    next := cntReg - 1.U
}
cntReg := next
```

A 4 Stage Shift Register



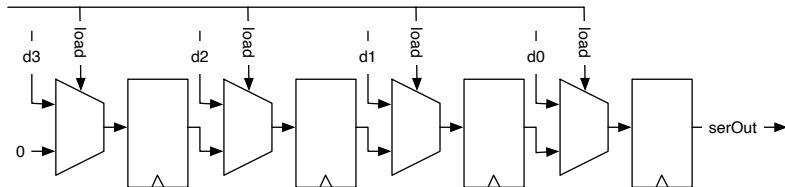
```
val shiftReg = Reg(UInt(4.W))  
shiftReg := shiftReg(2, 0) ## din  
val dout = shiftReg(3)
```

A Shift Register with Parallel Output



```
val outReg = RegInit(0.U(4.W))  
outReg := serIn ## outReg(3, 1)  
val q = outReg
```

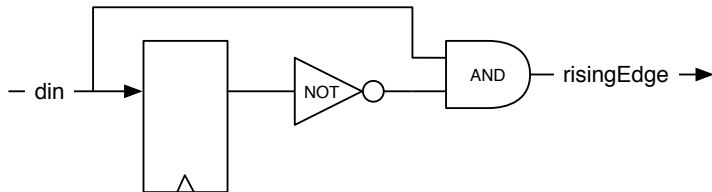
A Shift Register with Parallel Load



```
val loadReg = RegInit(0.U(4.W))
when (load) {
  loadReg := d
} otherwise {
  loadReg := 0.U ## loadReg(3, 1)
}
val serOut = loadReg(0)
```


A Simple Circuit

- ▶ What does the following circuit?
- ▶ Is this related to a finite-state machine?

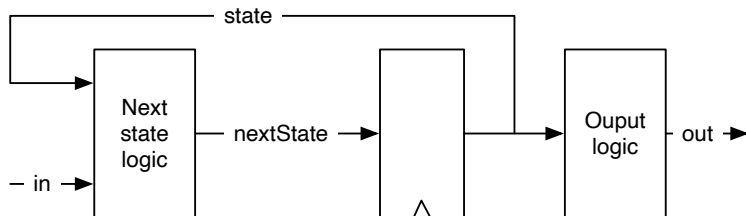


Finite-State Machine (FSM)

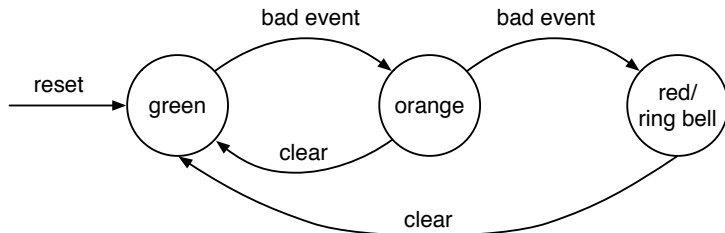
- ▶ Has a register that contains the state
- ▶ Has a function to computer the next state
 - ▶ Depending on current state and input
- ▶ Has an output depending on the state
 - ▶ And maybe on the input as well
- ▶ Every synchronous circuit can be considered a finite state machine
- ▶ However, sometimes the state space is a little bit too large

Basic Finite-State Machine

- ▶ A state register
- ▶ Two combinational blocks



State Diagram



- ▶ States and transitions depending on input values
- ▶ Example is a simple alarm FSM
- ▶ Nice visualization
- ▶ Will not work for large FSMs
- ▶ Complete code in the Chisel book

State Table for the Alarm FSM

State	Input		Next state	Ring bell
	Bad event	Clear		
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

The Input and Output of the Alarm FSM

- ▶ Two inputs and one output

```
val io = IO(new Bundle{  
    val badEvent = Input(Bool())  
    val clear = Input(Bool())  
    val ringBell = Output(Bool())  
})
```

Encoding the State

- ▶ We can optimize state encoding
- ▶ Two common encodings are: binary and one-hot
- ▶ We leave it to the synthesize tool
- ▶ Use symbolic names with an Enum
- ▶ Note the number of states in the Enum construct
- ▶ We use a Scala list with the `::` operator

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

Start the FSM

- ▶ We have a starting state on reset

```
val stateReg = RegInit(green)
```


The Next State Logic

```
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  
    }  
  }  
}
```

The Output Logic

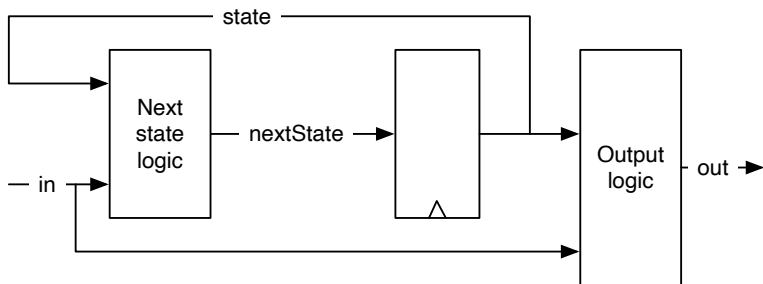
```
io.ringBell := stateReg === red
```

Summary on the Alarm Example

- ▶ Three elements:
 1. State register
 2. Next state logic
 3. Output logic
- ▶ This was a so-called Moore FSM
- ▶ There is also a FSM type called Mealy machine

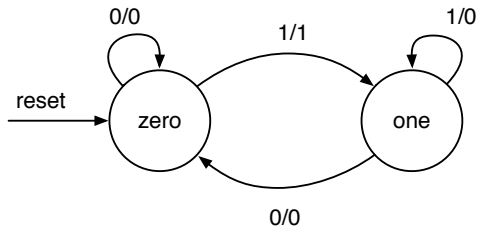
A so-called Mealy FSM

- ▶ Similar to the former FSM
- ▶ Output also depends in the input
- ▶ It is *faster*
- ▶ Less composable (draw it)



The Mealy FSM for the Rising Edge

- ▶ That was our starting example
- ▶ Output is also part of the transition arrows

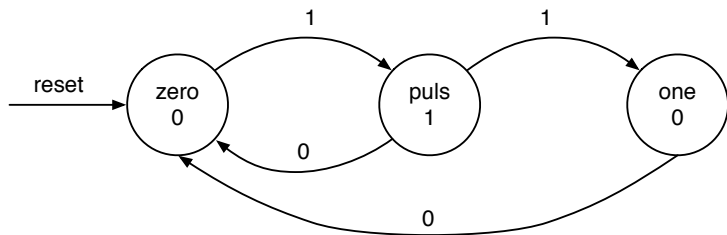


The Mealy Solution

- ▶ Show code from the book as it is too long for slides

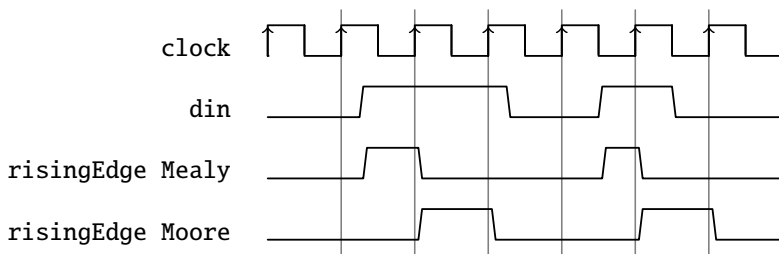
State Diagram for the Moore Rising Edge Detection

- We need three states



Comparing with a Timing Diagram

- Moore is delayed one clock cycle compared to Mealy



What is Better?

- ▶ It depends ;-)
- ▶ Moore is on the save side
- ▶ More is composable
- ▶ Mealy has *faster* reaction
- ▶ Both are tools in you toolbox
- ▶ Keep it simple with your vending machine and use a Moore FSM

Another Simple FSM

- ▶ a FSM for a single word buffer
- ▶ Just two symbols for the state machine

```
val empty :: full :: Nil = Enum(2)
```

Finite State Machine for a Buffer

```
val empty :: full :: Nil = Enum(2)
val stateReg = RegInit(empty)
val dataReg = RegInit(0.U(size.W))

when(stateReg === empty) {
  when(io.enq.write) {
    stateReg := full
    dataReg := io.enq.din
  }
}.elsewhen(stateReg === full) {
  when(io.deq.read) {
    stateReg := empty
  }
}
```

- A simple buffer for a bubble FIFO

A Collection of Signals with Vec

- ▶ Chisel Vec is a collection of signals of the same type
- ▶ The collection can be accessed by an index
- ▶ Similar to an array in other languages
- ▶ Wrap into a Wire() for combinational logic
- ▶ Wrap into a Reg() for a collection of registers

```
val v = Wire(Vec(3, UInt(4.W)))
```

Using a Vec

```
v(0) := 1.U
```

```
v(1) := 3.U
```

```
v(2) := 5.U
```

```
val idx = 1.U(2.W)
```

```
val a = v(idx)
```

- ▶ Reading from an Vec is a multiplexer
- ▶ We can put a Vec into a Reg

```
val registerFile = Reg(Vec(32, UInt(32.W)))
```

An element of that register file is accessed with an index and used as a normal register.

```
registerFile(idx) := dIn
```

```
val dOut = registerFile(idx)
```

Mixing Vecs and Bundles

- ▶ We can freely mix bundles and vectors
- ▶ When creating a vector with a bundle type, we need to pass a prototype for the vector fields. Using our `Channel`, which we defined above, we can create a vector of channels with:

```
val vecBundle = Wire(Vec(8, new Channel()))
```

- ▶ A bundle may as well contain a vector

```
class BundleVec extends Bundle {  
    val field = UInt(8.W)  
    val vector = Vec(4, UInt(8.W))  
}
```

Today's Lab

- ▶ This is the start of group work
 - ▶ Please register your group [here](#)
- ▶ Binary to 7-segment decoder
- ▶ First part of your vending machine
- ▶ Just a single digit, only combinational logic
- ▶ Use the nice tester provided to develop the circuit
- ▶ Then synthesize it for the FPGA
- ▶ Test with switches
- ▶ Show a TA your working design
- ▶ [Lab 6](#)

Summary

- ▶ Waveform testing is the way to develop/debug
- ▶ Counters are important tools, e.g., to generate timing
- ▶ Finite-state machines are another tool of the trade
- ▶ Two types: Moore and Mealy
- ▶ A Chisel Vec is the hardware version of an array