

Communicating State Machines

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

April 4, 2024

Overview

- ▶ Display multiplexing solution
- ▶ Ready/valid interface
- ▶ Serial interface (RS 232)
- ▶ A little bit of Scala
- ▶ Hardware generators

Exam Info

- ▶ Exam will be online
- ▶ All aids allowed, except Internet
- ▶ PDF with exam questions
- ▶ Upload solution in a single PDF
 - ▶ Please use your study number as file name
 - ▶ Train to do a drawing and integrate it into a PDF
- ▶ Timing exercise, some coding, understanding questions, drawing circuits
- ▶ I will upload older exams at DTU Learn and (partially) solutions

Group Workflow Suggestions

- ▶ Share code on GitHub (private repo)
- ▶ Meet in Zoom: you all have a full license from DTU
 - ▶ You can take over a screen to type
 - ▶ You can draw on it
 - ▶ Besides ad-hoc meetings, have regular project meetings
- ▶ Use Slack for quick notes and quick sharing of files
- ▶ Maybe also try to share the `.bit` file for the FPGA board
- ▶ Use Google docs for taking notes, start your report
- ▶ If you like Latex, use overleaf
- ▶ You can also work on the lab assignments at other times than the lab time ;-)

One Possible Solution for Last Lab

```
val MAX_CNT = 100000.U // use a smaller value  
    for waveform viewing
```

```
val tickCntReg = RegInit(0.U(32.W))  
val cntReg = RegInit(0.U(4.W))
```

```
val tick = tickCntReg === MAX_CNT  
tickCntReg := Mux(tick, 0.U, tickCntReg + 1.U)  
when (tick) {  
    cntReg := cntReg + 1.U  
}
```

```
val m = Module(new SevenSegDec())  
m.io.in := cntReg  
sevSeg := m.io.out
```

A Self-Running Tester

- ▶ DisplaySpec is a self-running circuit
- ▶ Has no input
- ▶ Needs (almost) no stimuli (poke)
- ▶ Just run for a few cycles

```
class DisplaySpec extends AnyFlatSpec with
  ChiselScalatestTester {
  "DisplayTest " should "pass" in {
    test(new
      Display(20)).withAnnotations(Seq(WriteVcdAnnotation)) {
      dut =>
      dut.io.sw.poke(0x1234.U)
      dut.clock.step(200)
    }
  }
}
```

Running the Test

- ▶ Does not really do any testing
- ▶ Just generated the waveform for debugging
- ▶ Just running 200 cycles does not show much
- ▶ Increase the number of running cycles to 100000000?
- ▶ Or use a different constant for testing?

Next Labs

- ▶ Next week: test a given Vending Machine (optional)
- ▶ The remaining weeks: work on the full Vending Machine

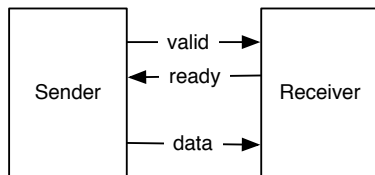
Communicating State Machines

- ▶ We did refactor a large FSM into smaller ones last week
- ▶ FSMs *communicate*
- ▶ Simple communication is:
 - ▶ Input processing to the FSM
 - ▶ FSM with the datapath
- ▶ More complex FSMs may exchange data with handshaking

Handshaking

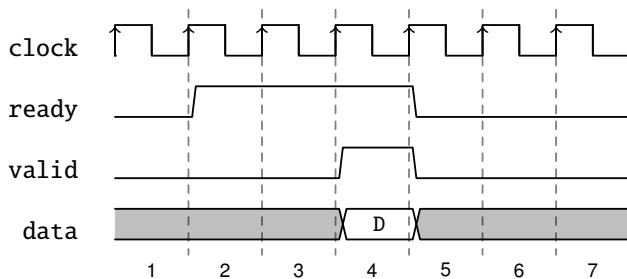
- ▶ Producer of data and consumer need to agree when data is transferred
- ▶ Producer tells when data is available/valid with a valid signal
- ▶ Consumer tells when it is ready to receive data with a ready signal
- ▶ When both are asserted the transfer takes place
- ▶ Also called *flow control*

Ready-Valid Interface

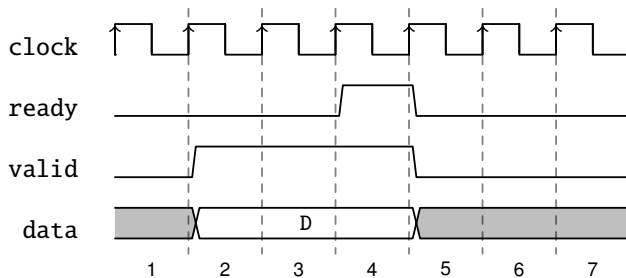


- ▶ Ready-valid flow control

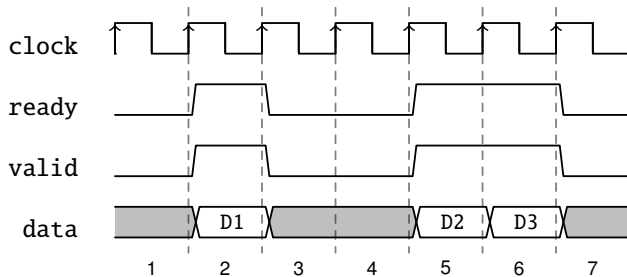
Ready-Valid Interface, Early Ready



Ready-Valid Interface, Late Ready



Single Cycle and Back-to-Back



Some Rules and Usage

- ▶ There shall be no combinational dependencies between `ready` and `valid`
- ▶ AXI uses `ready/valid` for all bus connections
- ▶ AXI restricts that `valid` asserted cannot be deasserted without a transaction
- ▶ AXI sender is not allowed to wait for `ready` before asserting `valid`
- ▶ AXI receivers do not have this restriction

Common Interface

- ▶ So common interface that Chisel defines a DecoupledIO

```
class DecoupledIO[T <: Data](gen: T) extends
  Bundle {
    val ready = Input(Bool())
    val valid = Output(Bool())
    val bits  = Output(gen)
  }
```


Serial I/O Interface

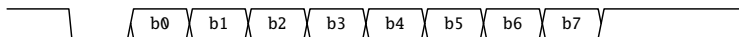
- ▶ Use only one wire for data transfer
 - ▶ Bits are serialized
 - ▶ That is where you need your shift register
- ▶ Shared wire or dedicated wires for transmit and receive
- ▶ Self timed
 - ▶ Serial UART (RS 232)
 - ▶ Ethernet
 - ▶ USB
- ▶ With a clock signal
 - ▶ SPI, I2C, ...

RS 232

- ▶ Old, but still common interface standard
 - ▶ Was common in 90' in PCs
 - ▶ Now substituted by USB
 - ▶ Still common in embedded systems
 - ▶ Your Basys 3 board has a RS 232 interface
- ▶ Standard defines
 - ▶ Electrical characteristics
 - ▶ '1' is negative voltage (-15 to -3 V)
 - ▶ '0' is positive voltage (+3 to +15 V)
 - ▶ Converted by a RS 232 driver to *normal* logic voltage levels

Serial Transmission

- ▶ Transmission consists of
 - ▶ Start bit (low)
 - ▶ 8 data bits
 - ▶ Stop bit(s) (high)
- ▶ Common baud rate is 115200 bits/s



RS 232 on the Basys 3

- ▶ You were asking for using more Basys 3 interfaces
- ▶ Basys 3 has an FTDI chip for the USB interface
- ▶ USB interface for FPGAs programming
- ▶ But also provides a RS 232 to the FPGA
- ▶ You can talk with your laptop
- ▶ Your VM could write out some text
- ▶ Open a terminal to watch (show it)
- ▶ Use Putty as terminal program

RS 232 from ip-contributions

- ▶ A collection of Chisel hardware components
- ▶ Contains the RS232/UART interface
- ▶ Uses the Decoupled interface
- ▶ Distributed as library from Maven Central
- ▶ No need to copy source around
- ▶ Just include it in your `build.sbt`
- ▶ Very easy distribution of open-source components
- ▶ You can contribute!
- ▶ [ip-contributions](#)

First Summary and a Break

- ▶ Main topic of today done
- ▶ Following is advanced material
- ▶ A little bit of Scala
- ▶ How to write hardware generators
- ▶ Maybe extend your display to show decimal number
- ▶ But first 10 minutes BREAK

Binary-Coded Decimal (BCD)

- ▶ Your current display shows numbers in hexadecimal
 - ▶ 15_{10} is displayed as $0F_{16}$
 - ▶ Which is in binary: 00001111
 - ▶ We would like to see it as a '1' followed by a '5'
 - ▶ Which is in binary: 0001 0101
- ▶ Convert from binary to binary-coded decimal (BCD)
 - ▶ But only for the display
 - ▶ Computing in BCD is hard

Binary to BCD Conversion Table

```
val bincode = io.sw(7,0)
val bcd = WireDefault(bincode)

switch(bincode) {
  is(0.U) { bcd := "b0000_0000".U }
  is(1.U) { bcd := "b0000_0001".U }
  is(2.U) { bcd := "b0000_0010".U }
  // ... some more
  is(9.U) { bcd := "b0000_1001".U }
  is(10.U) { bcd := "b0001_0000".U }
  is(11.U) { bcd := "b0001_0001".U }
  is(12.U) { bcd := "b0001_0010".U }
  // ... and many more entries
}

dispMux.io.price := bcd
```


Binary-Coded Decimal (BCD)

- ▶ Conversion is a table (= function)
- ▶ Combinational logic
- ▶ We *could* do the table manually
 - ▶ But it is large
 - ▶ The table has 100 entries to convert 0 to 99 to BCD
- ▶ Let's write a program for this
 - ▶ We could use Java, Python, TCL,...
 - ▶ With DE2 in VHDL I provided a Java program
 - ▶ Now we can do this directly in Chisel/Scala
- ▶ We will write our first *hardware generator*
- ▶ First we need a little bit of Scala

Chisel and Scala

- ▶ Chisel is a library written in Scala
 - ▶ Import the library with `import chisel3._`
- ▶ Chisel code is Scala code
- ▶ When it is run is *generates* hardware
 - ▶ Verilog for synthesize
 - ▶ Scala netlist for simulation (testing)
- ▶ Chisel is an embedded domain specific language
- ▶ Two languages in one can be a little bit confusing

Scala

- ▶ Is object oriented
- ▶ Is functional
- ▶ Strongly typed with very good type inference
- ▶ Runs on the Java virtual machine
- ▶ Can call Java libraries
- ▶ Consider it as Java++
 - ▶ Can almost be written like Java
 - ▶ With a more lightweight syntax

Scala Hello World

```
//- start hello_scala
object HelloScala extends App {
  println("Hello Chisel World!")
}
//- end
```

- ▶ Compile with `scalac` and run with `scala`
- ▶ You can even use Scala as a scripting language
- ▶ Or run with `sbt run`
- ▶ Show both

Scala Values and Variables

- ▶ Scala has two type of variables: vals and vars
- ▶ A val cannot be reassigned, it is a constant
- ▶ We use a val to name a hardware component in Chisel

```
// A value is a constant  
val zero = 0  
// No new assignment is possible  
// The following will not compile  
zero = 3
```

- ▶ Types are usually inferred
- ▶ But can be explicitly stated as follows

```
val number: Int = 42
```

Scala Variables

- ▶ A var can be reassigned, it is like a classic variable
- ▶ We use a var to write a hardware generator in Chisel

```
// We can change the value of a var variable  
var x = 2  
x = 3
```

Simple Loops

```
// Loops from 0 to 9  
// Automatically creates loop value i  
for (i <- 0 until 10) {  
  println(i)  
}
```

- ▶ We use a loop to generate hardware components

Scala for Loop for Circuit Generation

```
val regVec = Reg(Vec(8, UInt(1.W)))  
  
regVec(0) := io.din  
for (i <- 1 until 8) {  
  regVec(i) := regVec(i-1)  
}
```

- ▶ for is Scala
- ▶ This loop generates several connections
- ▶ The connections are parallel hardware
- ▶ This is a shift register

Conditions

```
for (i <- 0 until 10) {  
  print(i)  
  if (i%2 == 0) {  
    println(" is even")  
  } else {  
    println(" is odd")  
  }  
}
```

- ▶ Executed at runtime, when the circuit is created
- ▶ This is *not* a multiplexer

Scala Arrays and Lists

```
// An integer array with 10 elements  
val numbers = new Array[Int](10)  
for (i <- 0 until numbers.length) {  
  numbers(i) = i*10  
}  
println(numbers(9))
```

```
// List of integers  
val list = List(1, 2, 3)  
println(list)  
// Different form of list construction  
val listenum = 'a' :: 'b' :: 'c' :: Nil  
println(listenum)
```

Scala Classes

```
// A simple class
class Example {
  // A field, initialized in the constructor
  var n = 0

  // A setter method
  def set(v: Int) = {
    n = v
  }

  // Another method
  def print() = {
    println(n)
  }
}
```

Scala (Singleton) Object

```
object Example {}
```

- ▶ For *static* fields and methods
 - ▶ Scala has no static fields or methods like Java
- ▶ Needed for `main`
- ▶ Useful for helper functions

Singleton Object for the main

```
// A singleton object
object Example {

  // The start of a Scala program
  def main(args: Array[String]): Unit = {

    val e = new Example()
    e.print()
    e.set(42)
    e.print()
  }
}
```

- ▶ Compile and run it with sbt (or within Eclipse/IntelliJ):

```
sbt "runMain Example"
```

Conditional Circuit Generation

```
class Base extends Module { val io = new Bundle() }  
class VariantA extends Base { }  
class VariantB extends Base { }
```

```
val m = if (useA) Module(new VariantA())  
        else Module(new VariantB())
```

- ▶ if and else is Scala
- ▶ if is an expression that returns a value
 - ▶ Like “cond ? a : b;” in C and Java
- ▶ This is not a hardware multiplexer
- ▶ Decides which module to generate
- ▶ Could even read an XML file for the configuration

A Table with a Chisel Vec

- ▶ A Chisel `Vec` is a collection of signals/wires or registers
- ▶ Similar to an `Array` in other languages
- ▶ `Vec` in a `Wire` is a combinational table (multiplexer)
- ▶ `Vec` in a `Reg` is a collection of registers
- ▶ Create with number of elements and hardware type

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

A Combinational Vec

- ▶ A combinational Vec is basically a multiplexer
- ▶ Input signal/wire connected with a constant index
- ▶ Output select with a Chisel UInt signal

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

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

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

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

```
val a = v(index)
```

- ▶ Also convenient to represent a larger table
 - ▶ Instead of a *switch* table
 - ▶ Input can be *generated* with Scala code

Binary to BCD Conversion

```
import chisel3._

class BcdTable extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val table = Wire(Vec(100, UInt(8.W)))

  // Convert binary to BCD
  for (i <- 0 until 100) {
    table(i) := (((i/10)<<4) + i%10).U
  }

  io.data := table(io.address)
}
```

Today Lab

- ▶ Use of a UART
- ▶ [lab7](#)
- ▶ Is a communicating FSM problem, uses ready/valid handshake
- ▶ You can discuss your solution with a TA
- ▶ Could be used for extended functions in the Vending Machine
- ▶ You can now talk with your laptop :-)

Summary

- ▶ Communicating circuits/FSMs need handshaking
- ▶ Ready-valid interface
- ▶ Scala can be used to write circuit *generators*
- ▶ We explored generation of a binary to BCD conversion table