Communicating State Machines

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

April 8, 2021

Overview

- Repeat functions and parameters
- Input processing
- Ready/valid interface
- A little bit of Scala
- Hardware generators

Last (and Today's) Lab

- Display multiplexer
  - Input are 16 switches for 4 hexadecimal digits
- Almost 12/27 have already finished. This is GREAT!
- Please show it a TA when finished for a "tick"
- There is a simulation available, show it
- Maybe extend it with binary to BCD conversion
  - This is optional/advanced material
- You can also start to work on the full Vending Machine
  - Top-level, pin definition, and a simple tester are in:
  - `https://github.com/schoeberl/chisel-lab/tree/master/vending`

Next Labs

- Next week:
  - Almost 4 hours lab
  - Just vending machine specification recap
  - Maybe questions
- Work on the full Vending Machine

Functions

- Circuits can be encapsulated in functions
- Each *function call* generates hardware
- A function is defined with `def` *name*
- Similar to a method in Java
- Simple functions can be a single line

```
def adder(v1: UInt, v2: UInt) = v1 + v2

val add1 = adder(a, b)
val add2 = adder(c, d)
```

More Function Examples

- Functions can also contain registers

```
def addSub(add: Bool, a: UInt, b: UInt) =
  Mux(add, a + b, a - b)

val res = addSub(cond, a, b)

def rising(d: Bool) = d && !RegNext(d)

val edge = rising(cond)
```

The Counter as a Function

- Longer functions in curly brackets
- Last value is the return value

```
def counter(n: UInt) = {

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

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

val counter100 = counter(100.U)
```

Functional Abstraction

- Functions for repeated pieces of logic
- May contain state
- Functions may return *hardware*
- More lightweight than a `Module`

Parameterization

```
class ParamChannel(n: Int) extends Bundle {
  val data = Input(UInt(n.W))
  val ready = Output(Bool())
  val valid = Input(Bool())
}

val ch32 = new ParamChannel(32)
```

- Bundles and modules can be parametrized
- Pass a parameter in the constructor

A Module with a Parameter

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val c = Output(UInt(n.W))
  })

  io.c := io.a + io.b
}
```

- Parameter can also be a Chisel type

Use the Parameter

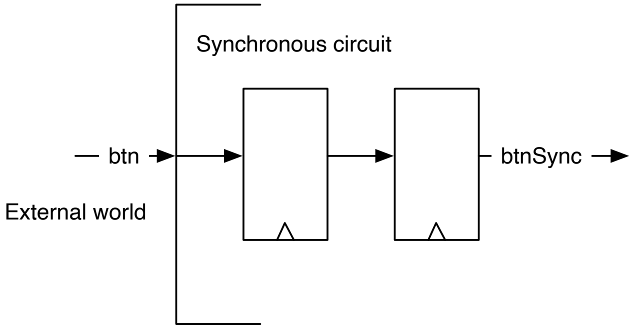
```
val add8 = Module(new ParamAdder(8))
val add16 = Module(new ParamAdder(16))
```

- Can be used for the display multiplexing configuration
- Switching between a constant for simulation and a constant for the FPGA

Input Processing

- Input signals are not synchronous to the clock
- May violate setup and hold time of a flip-flop
- Can lead to metastability
- Signals need to be *synchronized*
- Using two flip-flops
- Metastability cannot be avoided
- Assumption is:
  - First flip-flop may become metastable
  - But will resolve within the clock period

Input Synchronizer

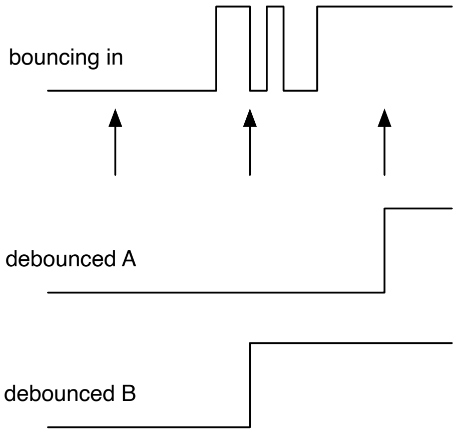


```
val btnSync = RegNext(RegNext(btn))
```

Bouncing Buttons

- Buttons and switches need some time to transition between on and off
- May bounce between the two values
- Without processing we detect more than one event
- Solution is to filter out bouncing
  - Can be done electrically (R + C + Schmitt trigger)
  - That is why you have the extra PCB with the buttons
  - But we can also do this digitally
- Assume bouncing time t_{bounce}
- Sample at a period $T > t_{bounce}$
- Only use sampled signal

Sampling for Debouncing



Sampling for Debouncing

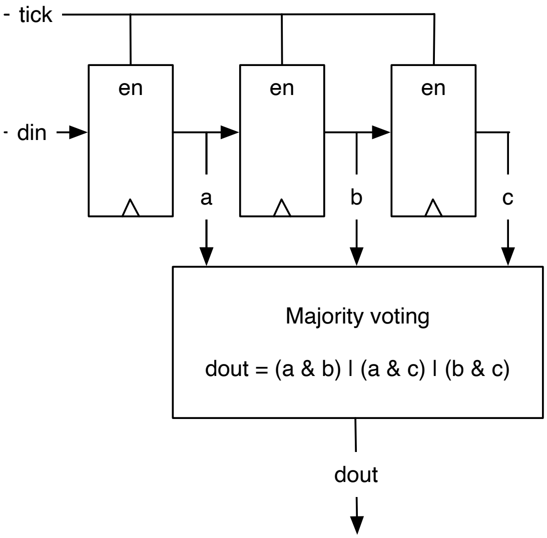
```
val FAC = 100000000/100

val btnDebReg = Reg(Bool())

val cntReg = RegInit(0.U(32.W))
val tick = cntReg === (FAC-1).U

cntReg := cntReg + 1.U
when (tick) {
  cntReg := 0.U
  btnDebReg := btnSync
}
```

- We already know how to do this!
- Just generate timing with a counter
- We sample at 100 Hz (bouncing below 10 ms)

Noisy Input Signal

- Sometimes input may be noisy
- May contain spikes
- Filtering with majority voting
- Majority voting of the sampled input signal
- However, this is seldom needed
- Not for he buttons

Majority Voting



Majority Voting

```
val shiftReg = RegInit(0.U(3.W))
when (tick) {
  // shift left and input in LSB
  shiftReg := Cat(shiftReg(1, 0), btnDebReg)
}
// Majority voiting
val btnClean = (shiftReg(2) & shiftReg(1)) |
   (shiftReg(2) & shiftReg(0)) | (shiftReg(1) &
   shiftReg(0))
```

Detecting the Press Event

- Edge detection
- You have seen this before
- Just to complete the input procssing

```
val risingEdge = btnClean & !RegNext(btnClean)

// Use the rising edge of the debounced and
// filtered button to count up
val reg = RegInit(0.U(8.W))
when (risingEdge) {
  reg := reg + 1.U
}
```

Combine Input Processing with Functions

- Using small functions to abstract the processing
- Debounce.scala

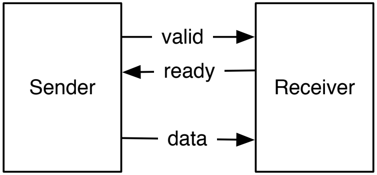
Communicating State Machines

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

Handshaking

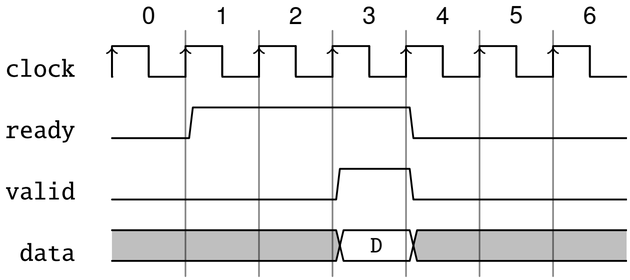
- Producer of data and consumer need to agree when data is transferred
- Producer tells when data is available/valid with a `valid` signal
- Consumer tells when it is ready toe receive data with a `ready` signal
- When both are asserted the transfer takes place
- Also called *flow control*

Ready-Valid Interface

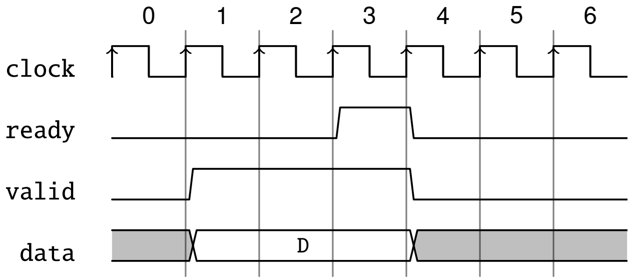


- Ready-valid flow control

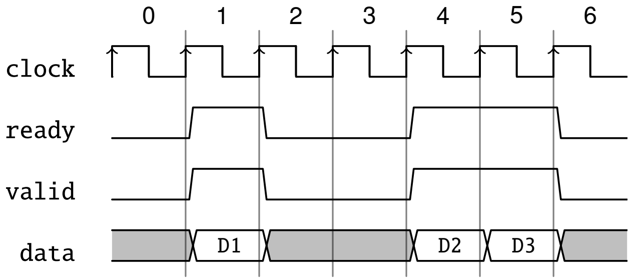
Ready-Valid Interface, Early Ready



Ready-Valid Interface, Late Ready



Single Cycle and Back-to-Back



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)
}
```

First Summary and 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,...
- 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

```scala
object HelloScala extends App{
  println("Hello Chisel World!")
}
```

- 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: `val`s and `var`s
- A `val` cannot be reassigned, it is a constant
- We use a `val` to name a hardware component in Chisel

```scala
// 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

```scala
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

```scala
// 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 shiftReg = RegInit(0.U(8.W))

shiftReg(0) := inVal

for (i <- 1 until 8) {
  shiftReg(i) := shiftReg(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) {
  if (i%2 == 0) {
    println(i + " is even")
  } else {
    println(i + " is odd")
  }
}
```

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

Scala Arrays and Lists

```scala
// 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

```scala
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 or registers
- Similar to an Array in other languages
- `Vec` in a `Wire` is a combinational table
- `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 idx = 1.U(2.W)
val a = v(idx)
```

- 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)
}
```

Summary

- External inputs need processing
- Minimum is a input synchronizer
- 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