

# Refactor of State Machines

Martin Schoeberl

Technical University of Denmark  
Embedded Systems Engineering

March 25, 2021

# Outline

- ▶ Today one hour lecture planned, but probably a bit longer
- ▶ Repeat finite-state machine with datapath
- ▶ Factoring of finite-state machines
- ▶ Advanced Chisel: functions and parametrization
- ▶ Talk on today's (and next week) lab exercise

# Exam Info

- ▶ Exam will be online at home
- ▶ All aids allowed with open Internet
- ▶ But DTU is very harsh on cheating, just don't do it
- ▶ Multiple choice plus PDF with 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
- ▶ We will do a test exam

## A (Minimal) Project from Scratch

- ▶ *It is cumbersome to set up new Chisel project without a template.*
- ▶ *I would like if we were taught how to create our own programs in Chisel, instead of just filling out the necessary code in a program you have already created.*
- ▶ Just two files: `build.sbt` and a `.scala` file
- ▶ Create the folder/directory structure
- ▶ Copy a `build.sbt`
- ▶ Import in IntelliJ
- ▶ Create a `.scala` class
- ▶ Show it now
- ▶ Do this exercise today!

# Workflow

- ▶ *It could be nice to hear more about how to best organize a project/workflow when you are multiple people working on the same project. Maybe some guiding principles and good practices that can be followed.*
- ▶ 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 a regular project meeting
- ▶ 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

# FSM with Datapath

- ▶ A type of computing machine
- ▶ Consists of a finite-state machine (FSM) and a datapath
- ▶ The FSM is the master (the controller) of the datapath
- ▶ The datapath has computing elements
  - ▶ E.g., adder, incrementer, constants, multiplexers, ...
- ▶ The datapath has storage elements (registers)
  - ▶ E.g., sum of money payed, count of something, ...

# FSM-Datapath Interaction

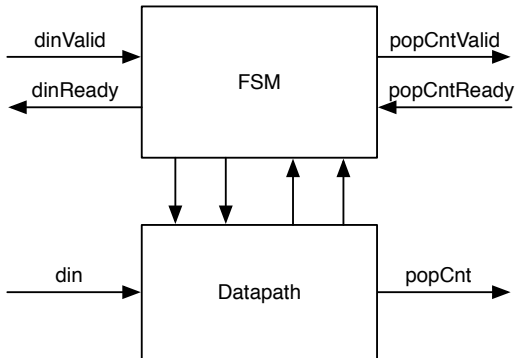
- ▶ The FSM controls the datapath
  - ▶ For example, add 2 to the sum
- ▶ By controlling multiplexers
  - ▶ For example, select how much to add
  - ▶ Not adding means selecting 0 to add
- ▶ Which value goes where
- ▶ The FSM logic also depends on datapath output
  - ▶ Is there enough money paid to release a can of soda?
- ▶ FSM and datapath interact

# Popcount Example

- ▶ An FSMD that computes the popcount
- ▶ Also called the Hamming weight
- ▶ Compute the number of '1's in a word
- ▶ Input is the data word
- ▶ Output is the count
- ▶ Code available at [PopCount.scala](#)

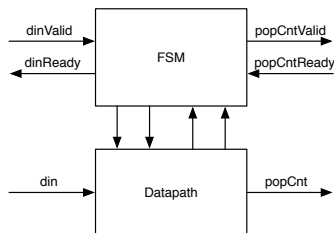


# Popcount Block Diagram



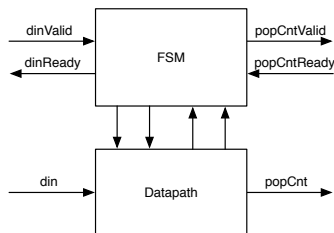
# Popcount Connection

- ▶ Input `din` and output `popCount`
- ▶ Both connected to the datapath
- ▶ We need some handshaking
- ▶ For data input and for count output

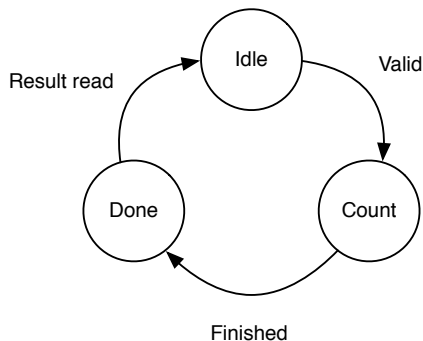


# Popcount Handshake

- ▶ We use a ready-valid handshake
- ▶ When data is available valid is asserted
- ▶ When the receiver can accept data ready is asserted
- ▶ Transfer takes place when both are asserted

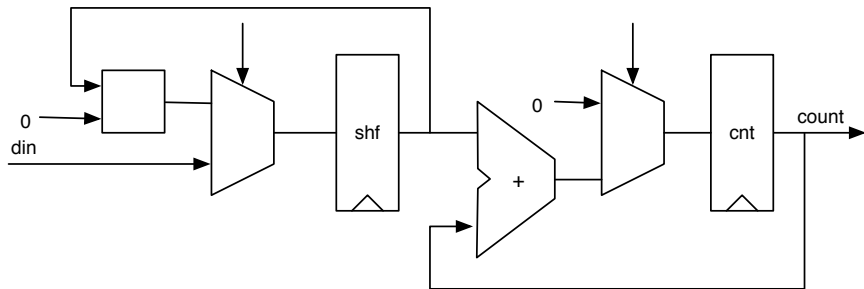


# The FSM



- ▶ A Very Simple FSM
- ▶ Two transitions depend on input/output handshake
- ▶ One transition on the datapath output

# The Datapath



# Let's Explore the Code

► In `PopCount.scala`

# Usage of an FSMD

- ▶ Maybe the main part your vending machine is an FSMD?

# Factoring FSMs

- ▶ Divide a big problem into several smaller problems
- ▶ Splitting a FSM into two or more
  - ▶ Simplify the design
- ▶ FSMs communicate via logic signals
  - ▶ FSM provides input controls signals to another
  - ▶ FSM senses output from another



# Specification Of a Light Flasher

- ▶ Inputs: start
- ▶ Outputs: light
- ▶ Operation:
  - ▶ When in = 1, FSM goes through 5 sequences:
    - ▶ On-Off-On-Off-On
  - ▶ Each On sequence (flash):
    - ▶ out = 1
    - ▶ 6 cycles long
  - ▶ Each Off sequence (space):
    - ▶ out = 0
    - ▶ 4 cycles long
  - ▶ After 5 sequences, FSM goes back to off state to wait for new input

# Light Flasher State Diagram

- ▶ Example from Dally, Chapter 17
- ▶ Copyright figure, so show it from older slides

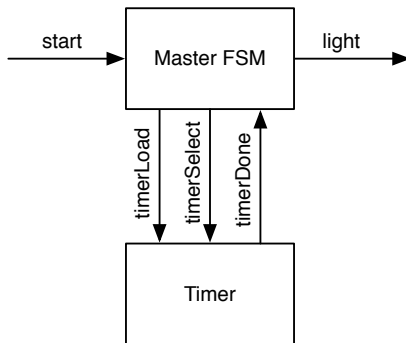
# Specification Change

- ▶ We have a flat FSM with 27 states
  - ▶ 27 `is(state)` statements
- ▶ If we change the specification to
  - ▶ 12 cycles for each flash
  - ▶ 4 flashes
  - ▶ 7 cycles between flashes
  - ▶ Complete change of `switch` statement
  - ▶ Now 70 `is` statements!
- ▶ This does not scale

# Factor Light Flasher

- ▶ Factor out counting on and off intervals
  - ▶ Into a timer
  - ▶ Reduces 6 and 4 states sequences into two single states
- ▶ Results in
  - ▶ a master FSM and
  - ▶ a timer FSM
- ▶ Simplifies FSMs
- ▶ Allows easier change of interval lengths

# Factored Light Flasher



- ▶ Time loads value 5 or 3, based in `timerSelect`

# Timer Specification

- ▶ Two inputs
  - ▶ `timerLoad` to load the down counter
  - ▶ `timerSelect` to select between 6 and 4 cycles counting
- ▶ Output
  - ▶ `timerDone` is 1 when counter has completed the countdown
  - ▶ Remains asserted until counter reloaded
- ▶ Counter can be (re)loaded in any state
  - ▶ When not loaded it counts down to zero
- ▶ Similar to the timer we looked at two weeks ago

# The Timer FSM

```
val timerReg = RegInit(0.U)
timerDone := timerReg === 0.U

// Timer FSM (down counter)
when(!timerDone) {
  timerReg := timerReg - 1.U
}
when (timerLoad) {
  when (timerSelect) {
    timerReg := 5.U
  } .otherwise {
    timerReg := 3.U
  }
}
```

# The Master FSM

- ▶ Show in IntelliJ
- ▶ Run test and show waveform



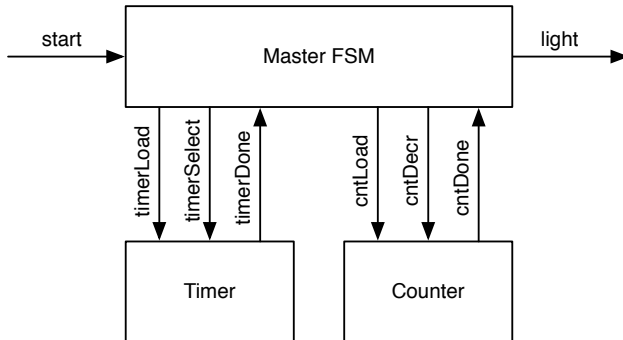
# Result of Refactoring

- ▶ State of original flat FSM has been separated
- ▶ The part of cycle counting in the counter
- ▶ Part flash or space in master FSM
- ▶ Represent original 27 states in just two 6 states FSMs
- ▶
- ▶ BTW: the master FSM is a Mealy FSM

## Still Redundancy in FSM

- ▶ flash1, flash2, and flash3 same function
- ▶ space1 and space2 same function
- ▶ Refactor number of remaining flashes
- ▶ Master FSM states: off, flash, and space

## Factor out “flash number”



# Counter

```
val cntReg = RegInit(0.U)
cntDone := cntReg === 0.U

// Down counter FSM
when(cntLoad) { cntReg := 2.U }
when(cntDecr) { cntReg := cntReg - 1.U }
```

- ▶ Loaded with 2 for 3 flashes
- ▶ Counts the *remaining* flashes

# Code of Flasher2

- ▶ Show in IntelliJ
- ▶ Run test and show waveform

# Benefits of Refactored Solution

- ▶ Master FSM has just three states: `off`, `flash`, and `space`
- ▶ Change of intervals or number of flashes needs no change in the FSM
- ▶ Smaller components are easier to read and simpler to test individually

# Usage in your VM

- ▶ Maybe factor out the edge detection for the button(s)
- ▶ Use a timer for more advanced user interface
  - ▶ Blinking LED on some error
  - ▶ Write text as a banner in the 7-segment display
  - ▶ ...

# 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

# Today Lab

- ▶ Do a project from scratch
- ▶ Display multiplexing
- ▶ Described in Vending Machine Specification (show)
- ▶ This week and next week
- ▶ Can also be developed in simulation

# Display Multiplexing

- ▶ Saving of pins in the FPGA
- ▶ Switch between the four digits at around 1 kHz
- ▶ Switch *faster* in simulation
- ▶ Show code
- ▶ Also includes a display simulation for those without an FPGA
- ▶ [Lab 8](#)



# Summary

- ▶ Divide a bigger problem into smaller ones
  - ▶ Easier to design
  - ▶ Easier to test
  - ▶ Sometimes only feasible solution
- ▶ Factoring state machines
  - ▶ Separate state into multiple 'orthogonal' state variables
  - ▶ Each is simpler to handle (fewer states)
  - ▶ "Factors out" repetitive sequences
  - ▶ Hierarchical structure