Finite State Machine with Datapath

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

March 14, 2024

Overview

- Review `Vec`
- Counter based circuits
- Finite-state machines (FSMs)
- FSM with Datapath
- Input processing

Last Lab

- A table to describe a 7-segment decoder and drive it
- Did you finish the exercises?
- Did you run it on your Basys3 board?
- Show it a TA for a tick!

TinyTapeout

- Do a *real* chip within DE 2
- a multi-project waver (within a multi-project waver)
- I can pay a few projects
- TinyTapeout
- Chisel template

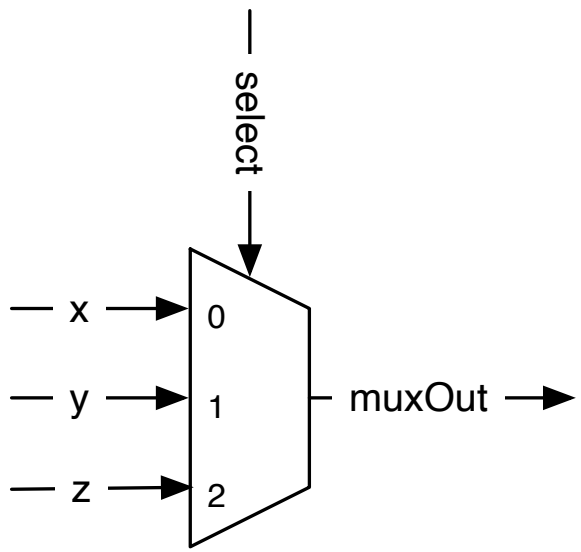
Vectors

- A powerful abstraction
- Let us repeat it today
- A vector (`Vec`) is an indexable collection
- Similar to an array in Java
- Selecting an element for read is a multiplexer
- Selecting an element to write is an input to a multiplexer or a register enable
- Bundles are constructs to structure data
- Similar to a class in Java or a record in C/VHDL

A Vector is a Multiplexer

- Follwing code is a 3:1 multiplexer

```
val m = Wire(Vec(3, UInt(8.W)))
m(0) := x
m(1) := y
m(2) := z
val muxOut = m(select)
```



A Vector of Registers

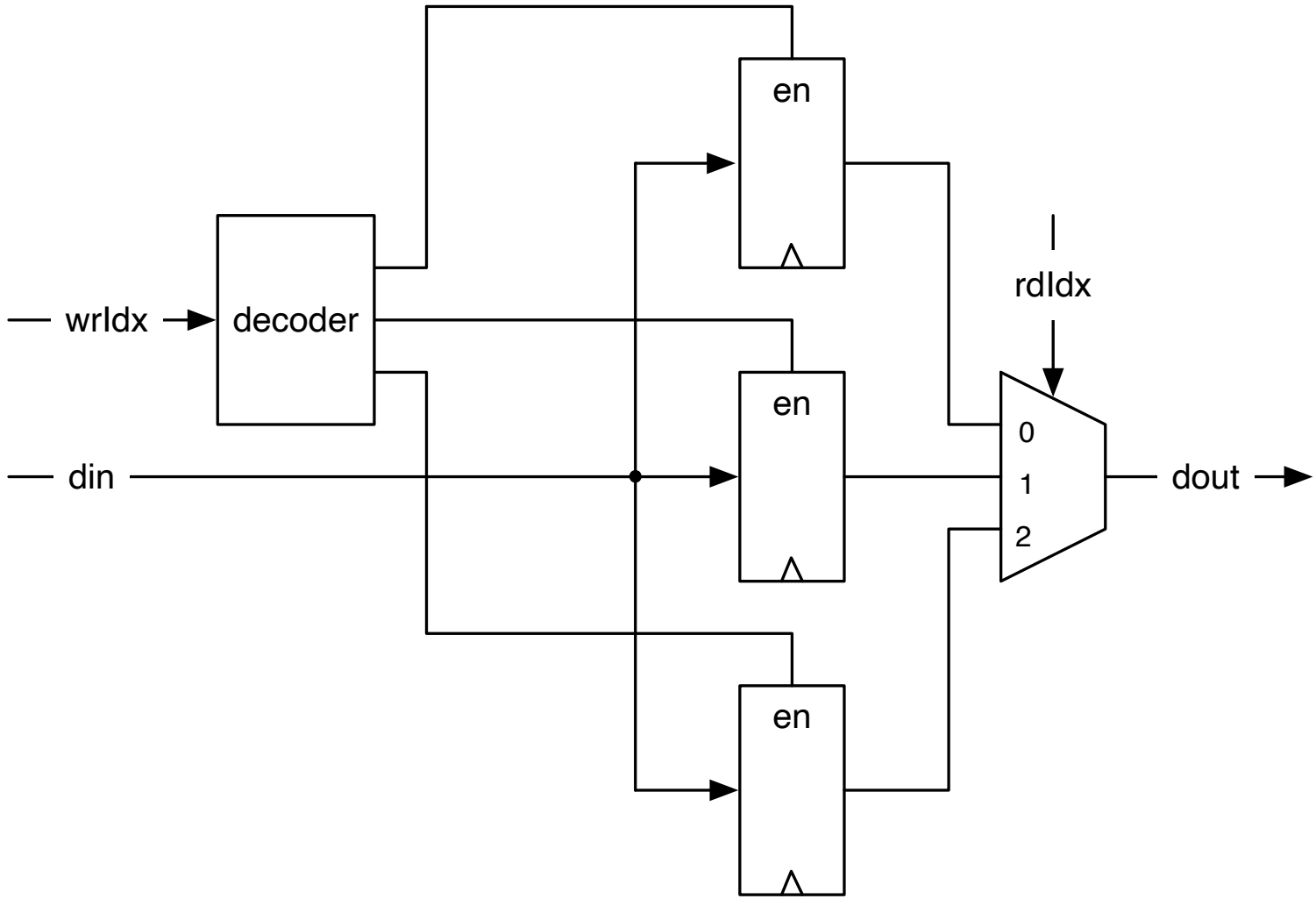
- Following code shows vectors and registers in action

```
val vReg = Reg(Vec(3, UInt(8.W)))

val dout = vReg(rdIdx)
vReg(wrIdx) := din
```

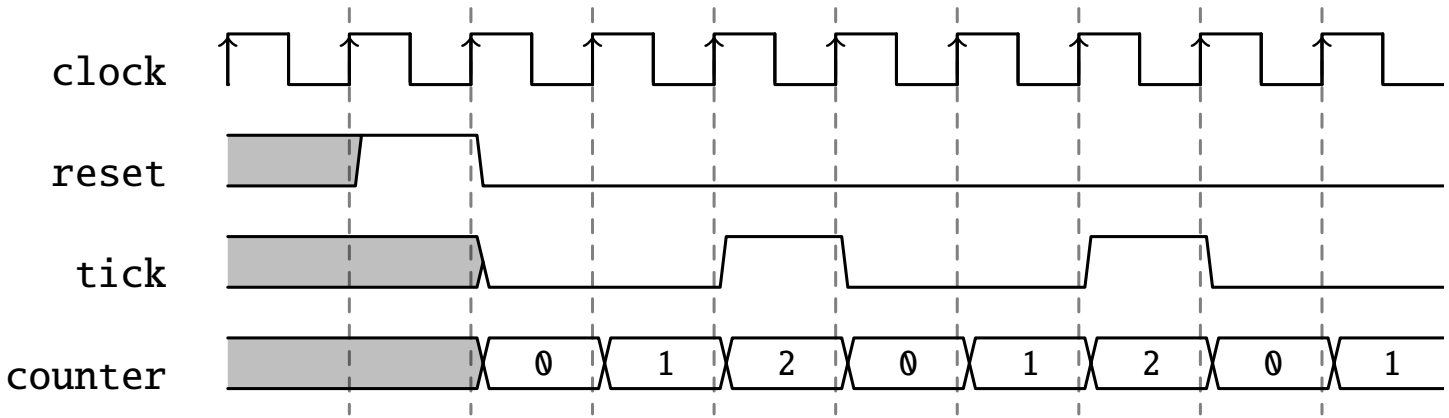
- Can you draw the schematic?

Schematic of the Reg of Vec



Generating Timing with Counters

- Generate a `tick` at a lower frequency
- We used it in Lab 1 for the blinking LED
- You needed it for last week's lab
- We will use it again in next week's lab
- 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
}
```

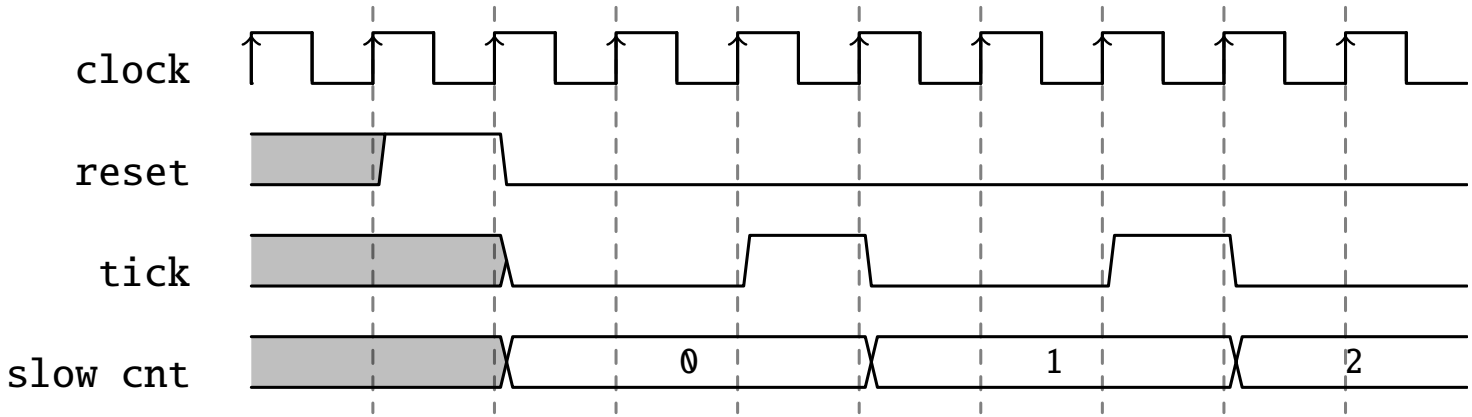
Using the Tick

- 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`



What is the Use of This *Slow* Counter?

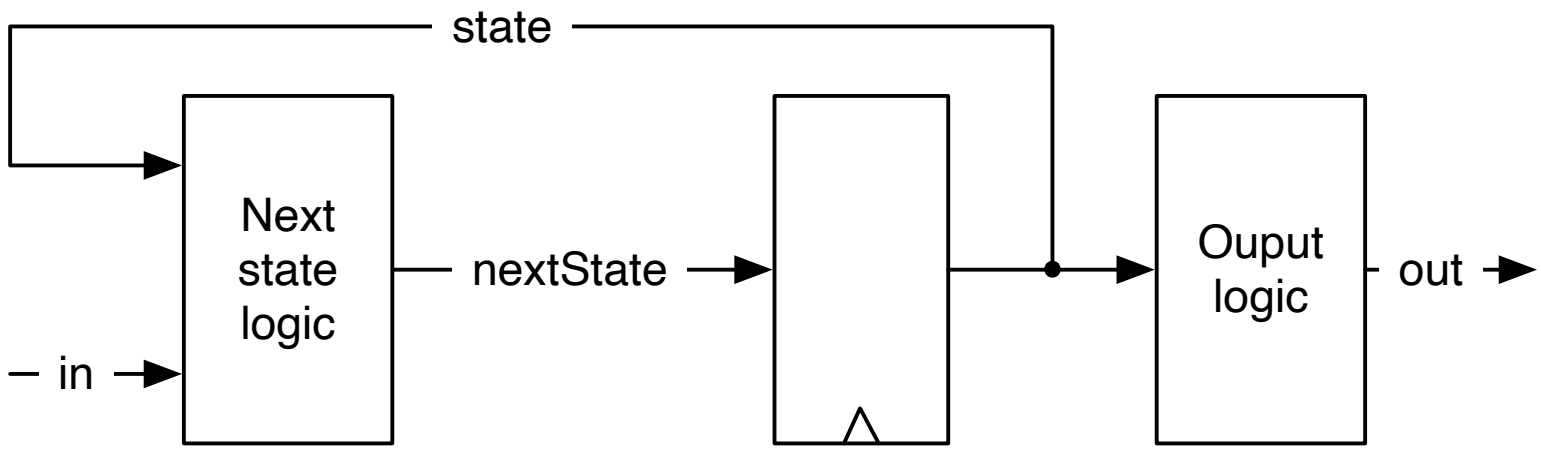
- This will be your lab exercise next week
- For the display multiplexing
- Then you need to generate a timing of 1 kHz (1 ms)

Finite-State Machine (FSM)

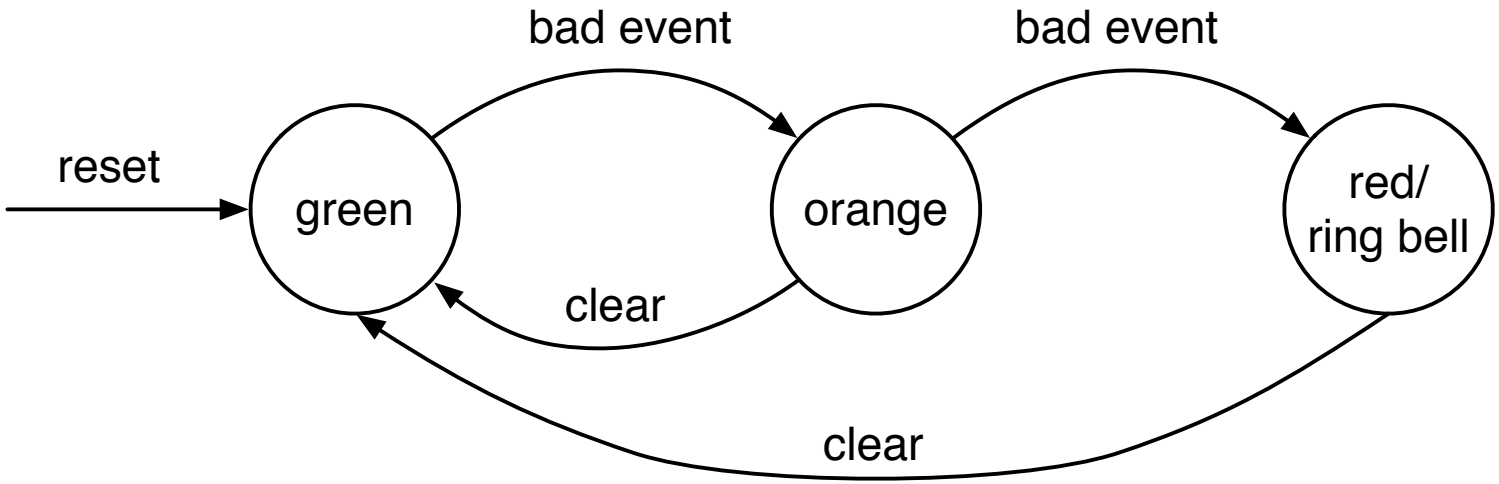
- Has a register that contains the state
- Has a function to computer the next state
  - Depending on current state and input
- Has an output depending on the state
  - And maybe on the input as well
- Every synchronous circuit can be considered a finite state machine
- However, sometimes the state space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

Basic Finite-State Machine

- A state register
- Two combinational blocks
  - Next state logic
  - Output logic



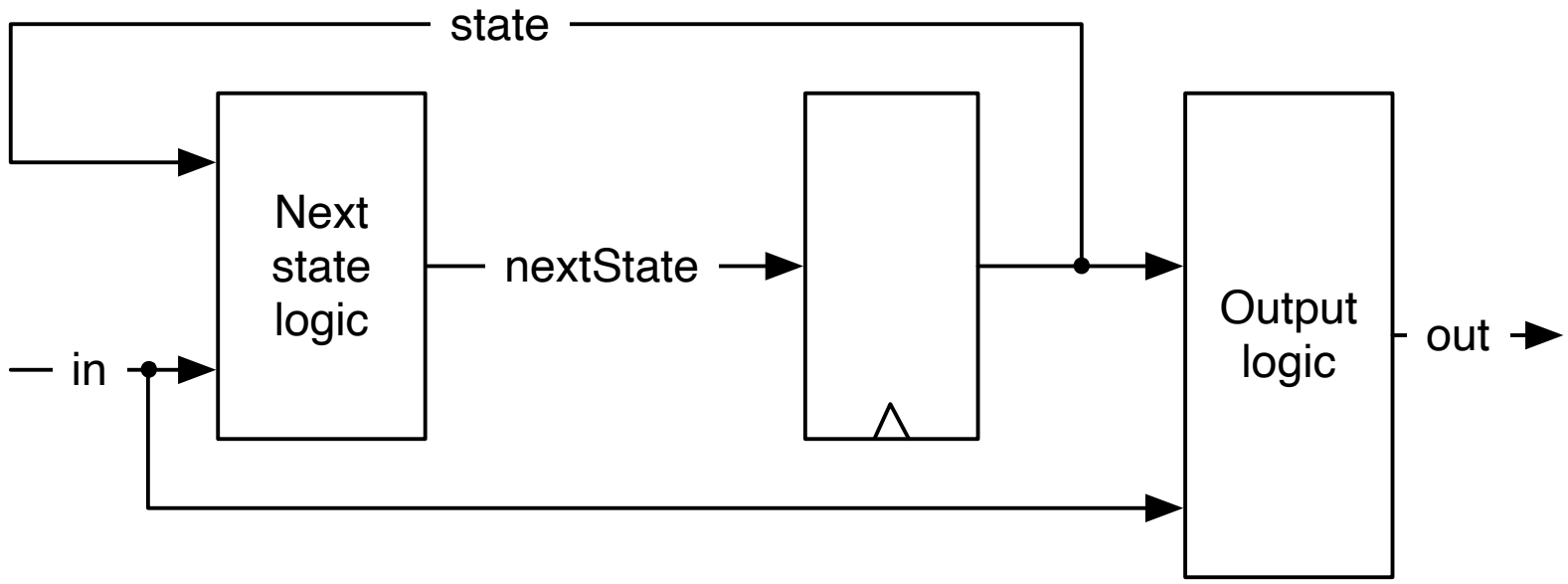
State Diagrams are Convenient



- States and transitions depending on input values
- Example is a simple alarm FSM
- Nice visualization
- Will not work for large FSMs

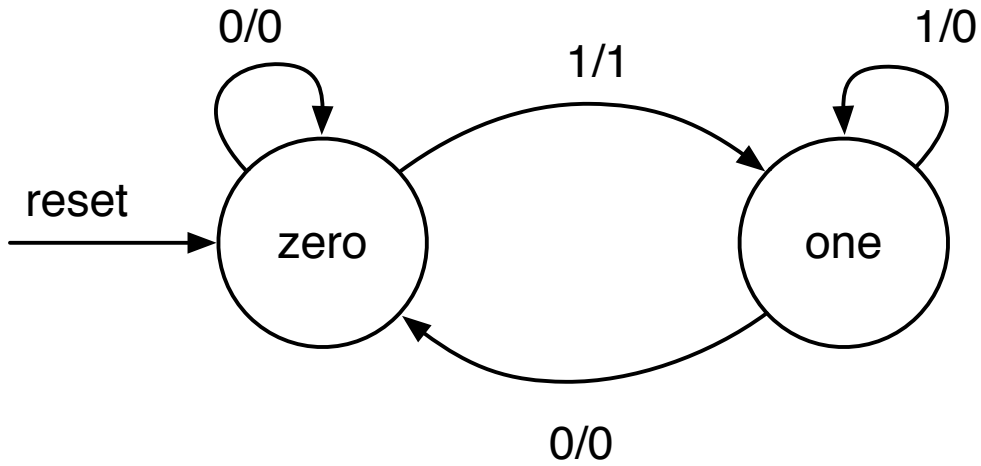
A Mealy FSM

- Similar to the former FSM
- Output also depends in the input
- Output is *faster*
- Less composable as we may have combinational circles



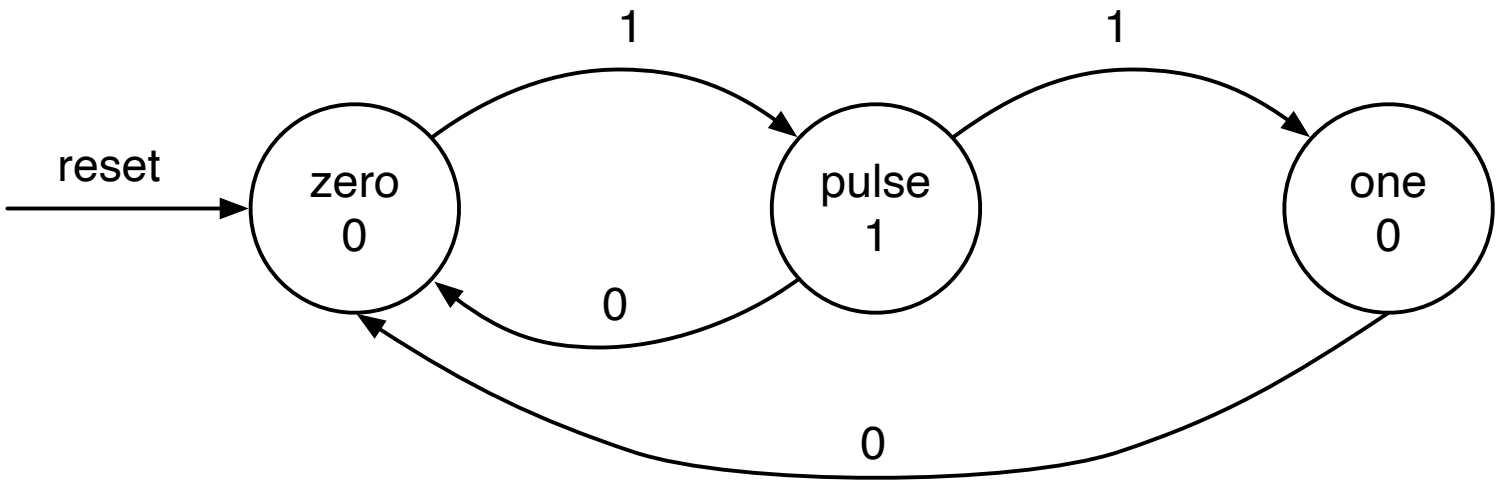
The Mealy FSM for the Rising Edge

- Output is also part of the transition arrows



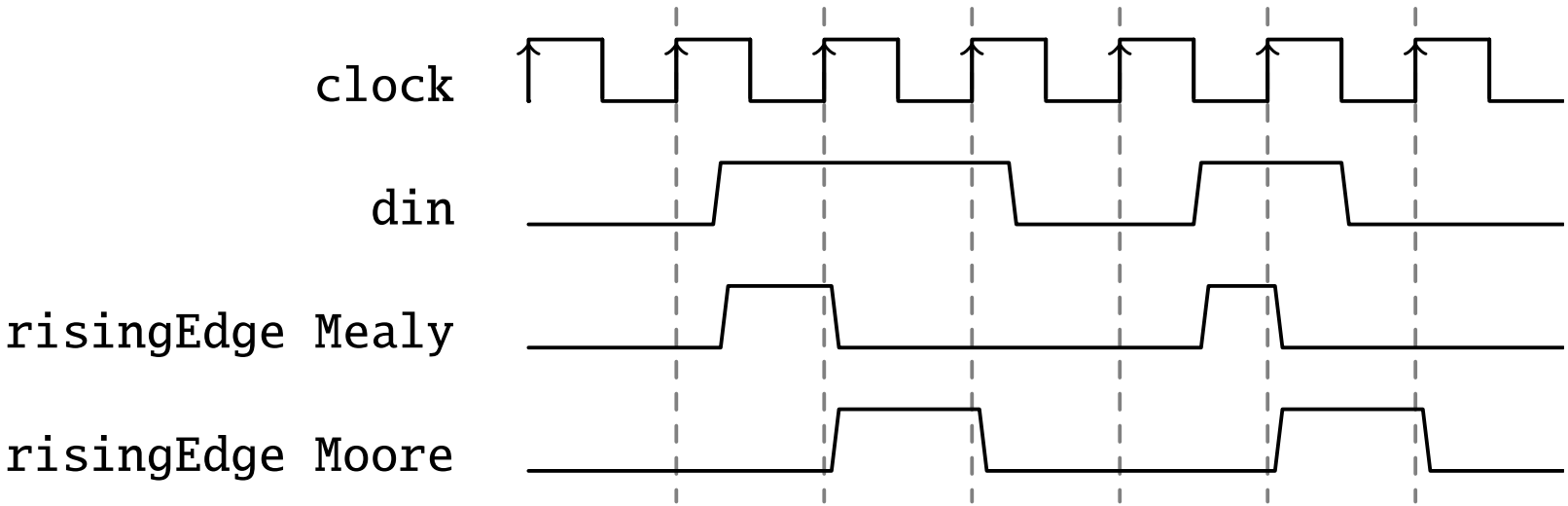
State Diagram for the Moore Rising Edge Detection

- We need three states



Comparing with a Timing Diagram

- Moore is delayed by one clock cycle compared to Mealy



What is Better?

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

Working Break

- 20’ break
- We are half way through the course
- Therefore, a midterm evaluation
- Send also on Slack (from the slide sources or the website)

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, ...

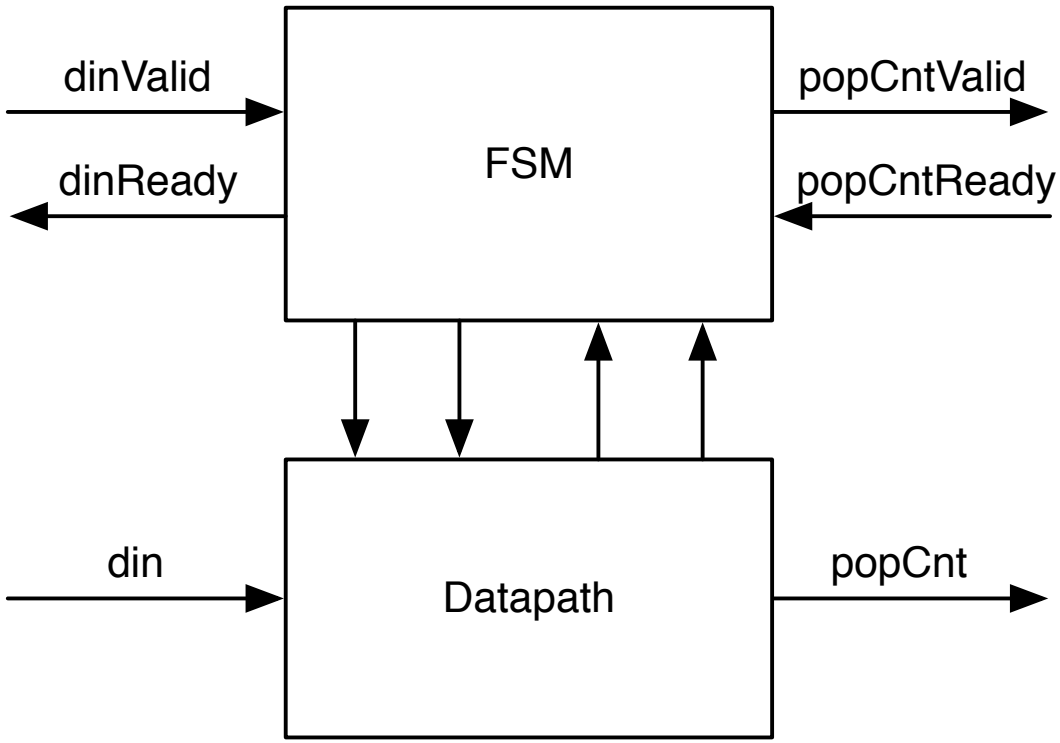
FSM-Datapath Interaction

- 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 payed to release a can of soda?
- FSM and datapath interact

Popcount Example

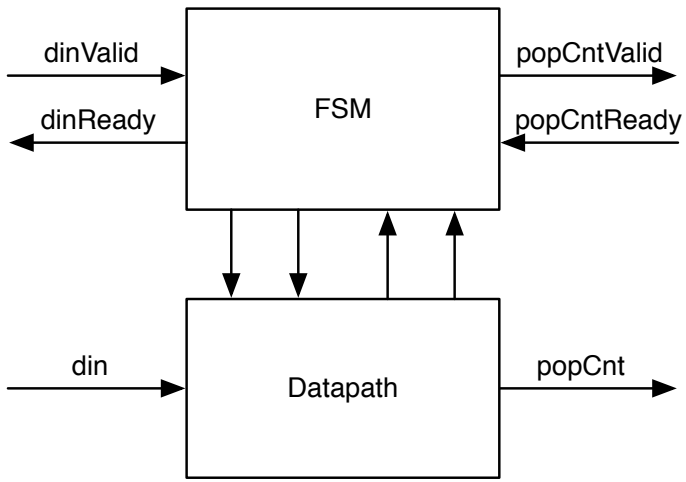
- An FSMD that computes the popcount
- Also called the Hamming weight
- Compute the number of ‘1’s in a word
- Input is the data word
- Output is the count
- Code available at PopulationCount.scala

Popcount Block Diagram



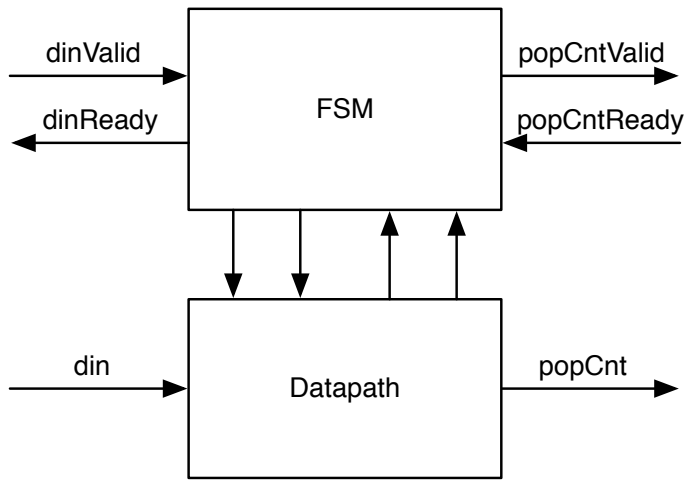
Popcount Connection

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

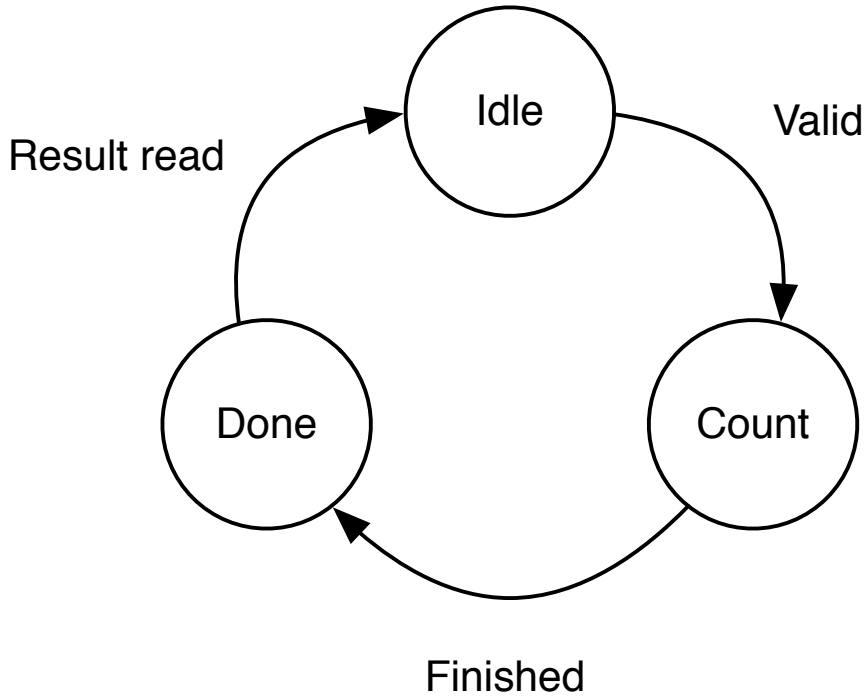


Popcount Handshake

- 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
- Draw the ready/valid handshake on the black board

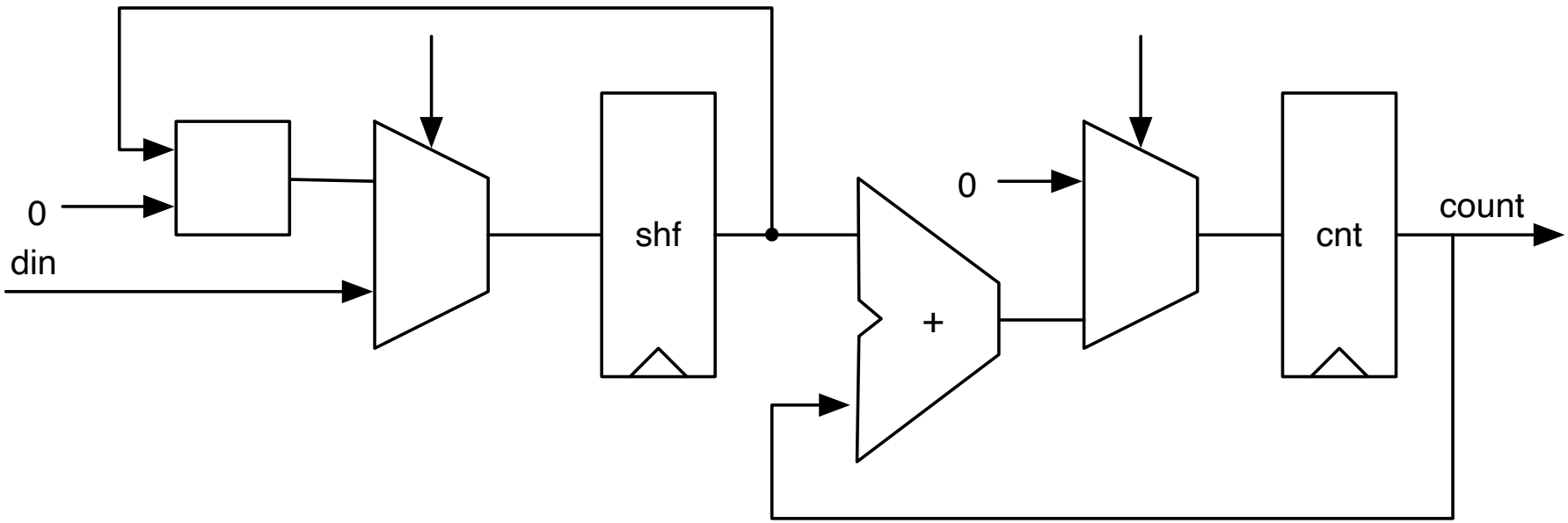


The FSM



- 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 PopulationCount.scala

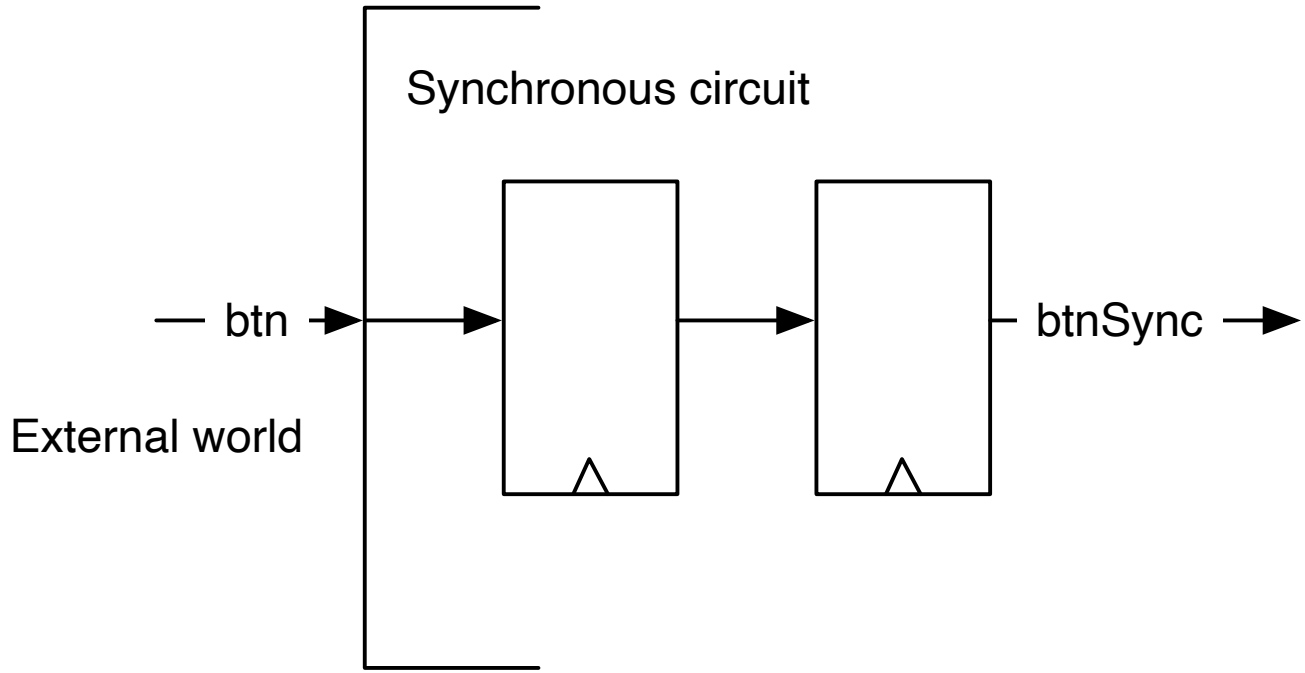
Usage of an FSMD

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

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 can arrive at different clock cycles at different places

Input Synchronizer

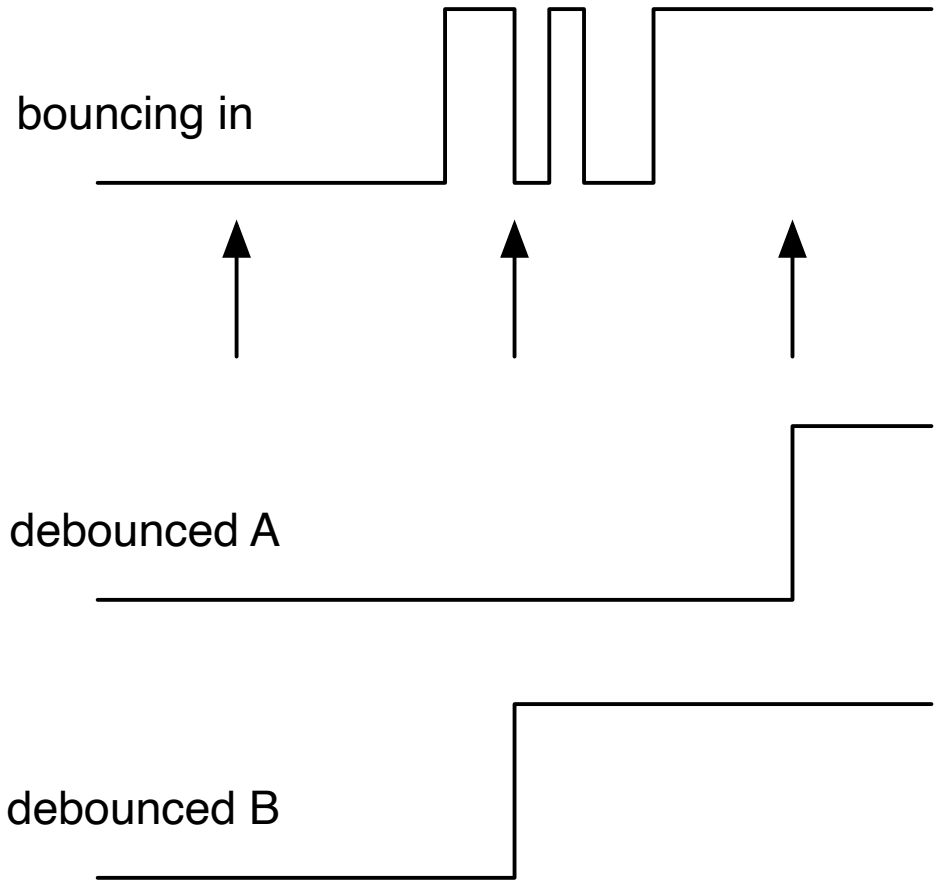



```
val btnSync = RegNext(RegNext(btn))
```

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
  - You can then drop your additional board ;-)
- 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)

Agile Hardware Design Course

- Advanced Chisel
- by Scott Beamer from UC Santa Cruz
- Includes executable slides
- https://classes.soe.ucsc.edu/cse228a/Winter23/
- Includes Videos
- I will do a similar course for the new CE Bachelor

Today's Lab

- Paper & pencil exercises (see course website)
- Exercises on FSMs
- From the Dally book
- Just sketch the Chisel code
- On paper or in a plain text editor
- As usual, show and discuss your solution with a TA
- Also finish you lab from last week and get the tick

Summary

- Counters are used to generate timing
- An FSM can control a datapath, which is an FSMD
- An FSMD is a computing machine
- Input needs to be processed (synchronize, maybe debounce)