Introduction to Chisel

A collection of slides by Martin Schoeberl

Assembled and presented by Luca Pezzarossa

Technical University of Denmark
Embedded Systems Engineering

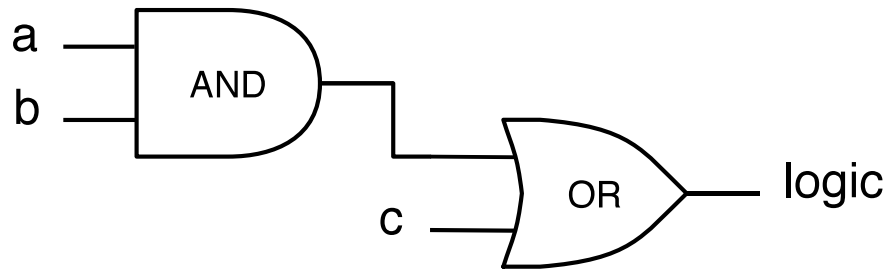
Overview

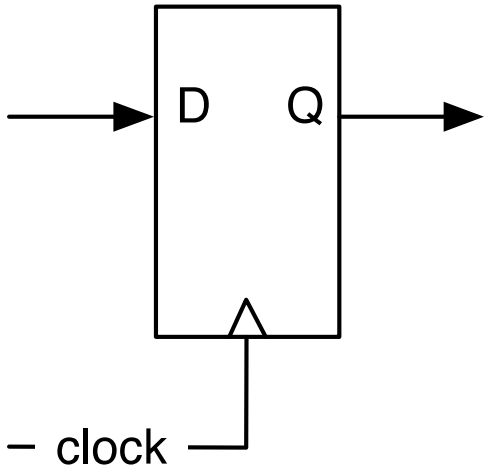
- Introduction
- Types
- Operators
- Module
- Combinational Circuits
- Sequential Circuits (Using Registers)
- Collections
- Connecting Modules
- Testing and Debugging
- Finite State Machines (with Datapath)

Introduction

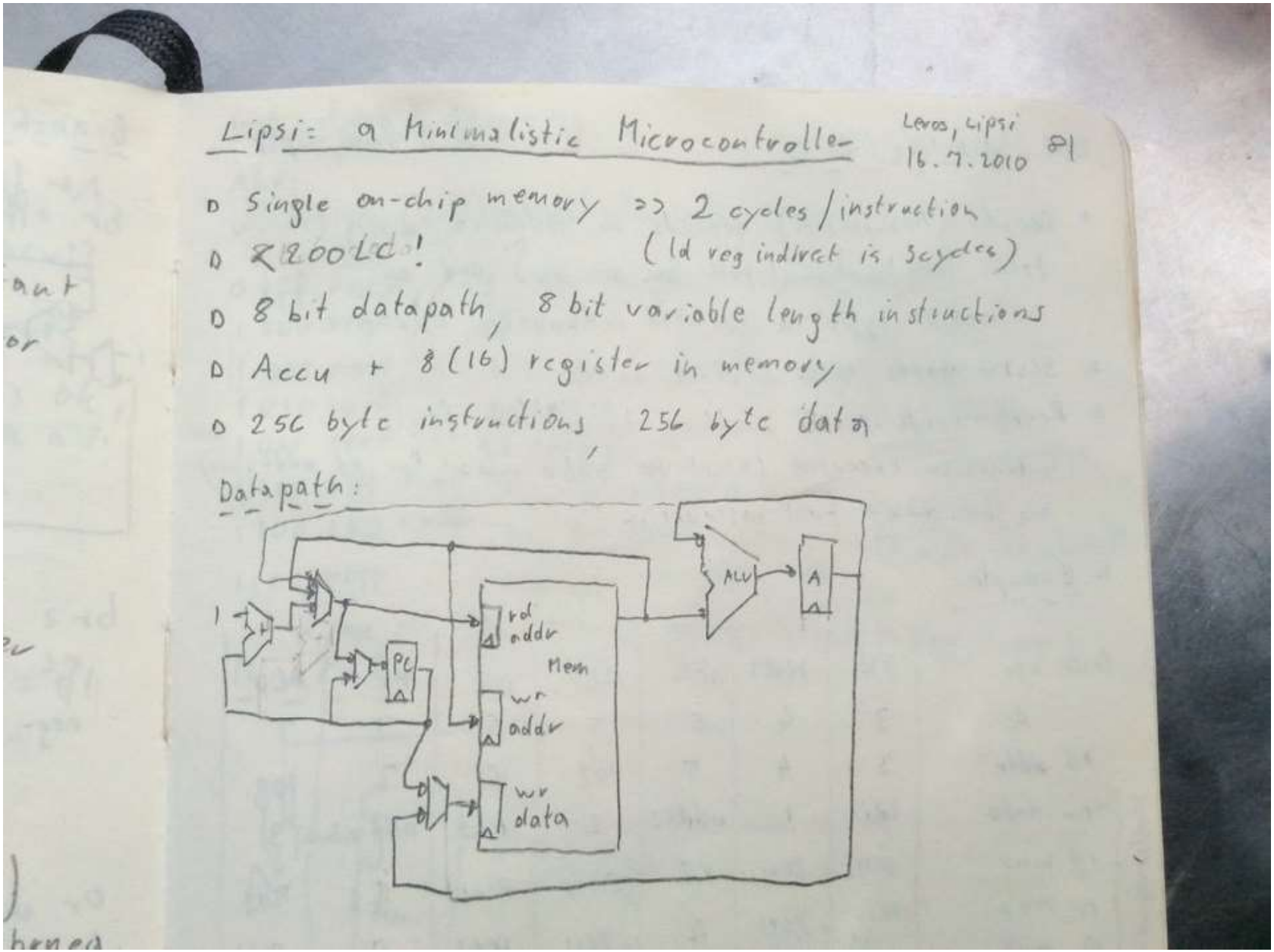
The Digital Abstraction

- Just two values: 0 and 1, or low and hight
- Represented as voltage
- Digital signals tolerate noise
- Digital Systems are *simple*, just:
  - Combinational circuits and
  - Registers





All Hardware Designs Start with a Diagram



Chisel

- A hardware *construction* language
  - Constructing Hardware In a Scala Embedded Language
  - If it compiles, it is synthesisable hardware
- Chisel is not a high-level synthesis language
- Single source for two targets
  - Cycle accurate simulation (testing)
  - Verilog for synthesis
- Embedded in Scala
  - Full power of Scala available
  - But to start with, no Scala knowledge needed
- Developed at UC Berkeley

Why Chisel Instead of VHDL/Verilog/SystemVerilog?

- Company X does Verilog, company Y does VHDL
  - Why Chisel?
- We learn principles of digital design, not tools
  - We pick a language that is modern and productive
- When knowing principles, switching the language is a matter of a week
- You are the future engineers and shall learn new tools
- You may then bring Chisel into the company

Chisel Success Stories

- Chisel Conference 2020 in silicon valley
- 90 participants
- More than 30 different chip companies present
- Several companies are looking into Chisel
- IBM did an open-source PowerPC
- SiFive is a RISC-V startup success
  - High productivity with Chisel
  - Open-source Rocket chip
- Esperanto uses the BOOM processor in Chisel
- Google did a machine learning processor
- Intel is looking at Chisel
- Chisel is open-source, if there is a bug you can fix it

Chisel vs. Scala

- A Chisel hardware description is a Scala program
- Chisel is a Scala library
- When the program is executed it generates hardware
- Chisel is a so-called *embedded domain-specific language*

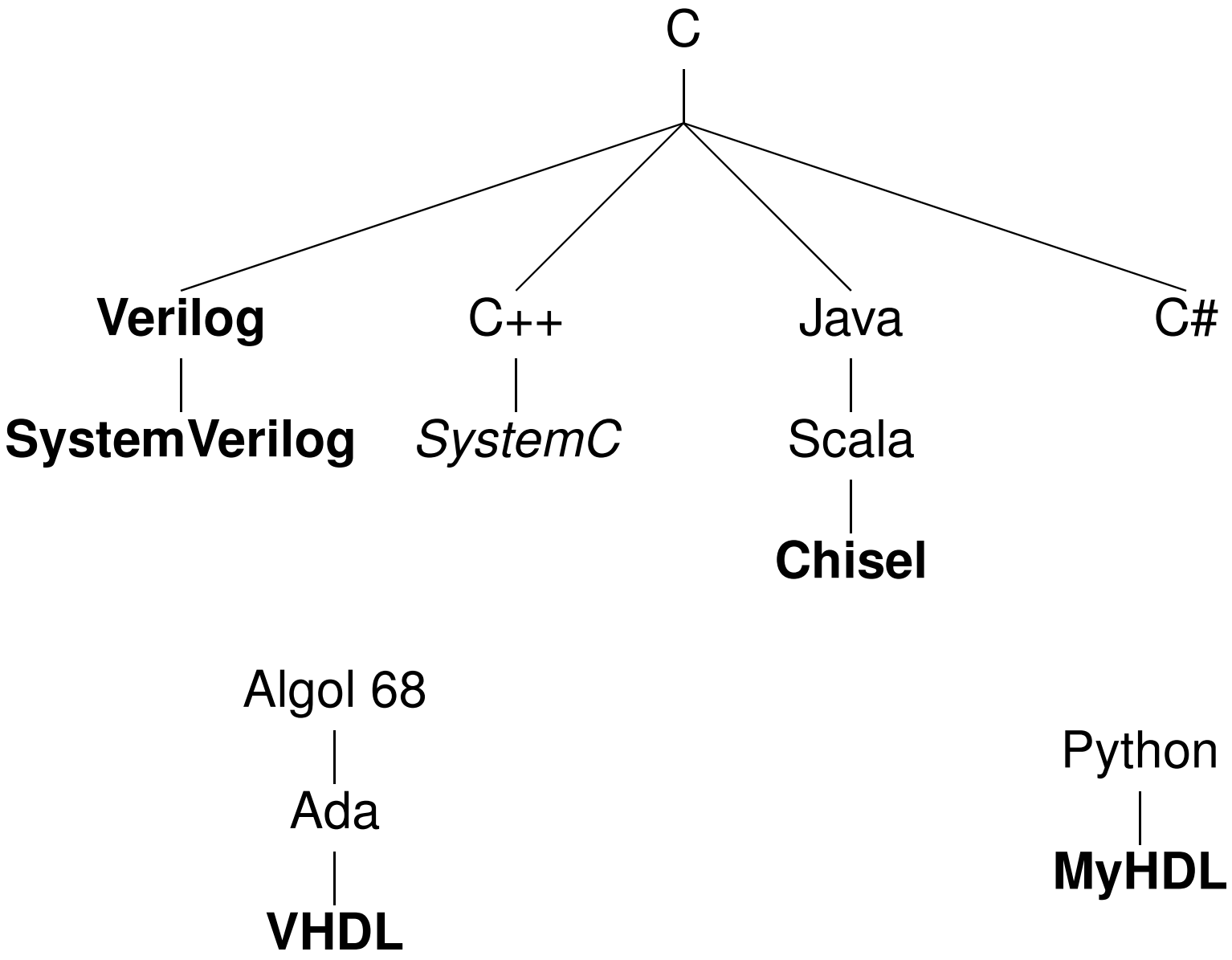
Some Notes on Scala

- Object oriented
- Functional
- Strongly typed
- Could be seen as Java++
- Compiled to the JVM
- Good Java interoperability
  - Many libraries available
  - You can write your testing code in Java

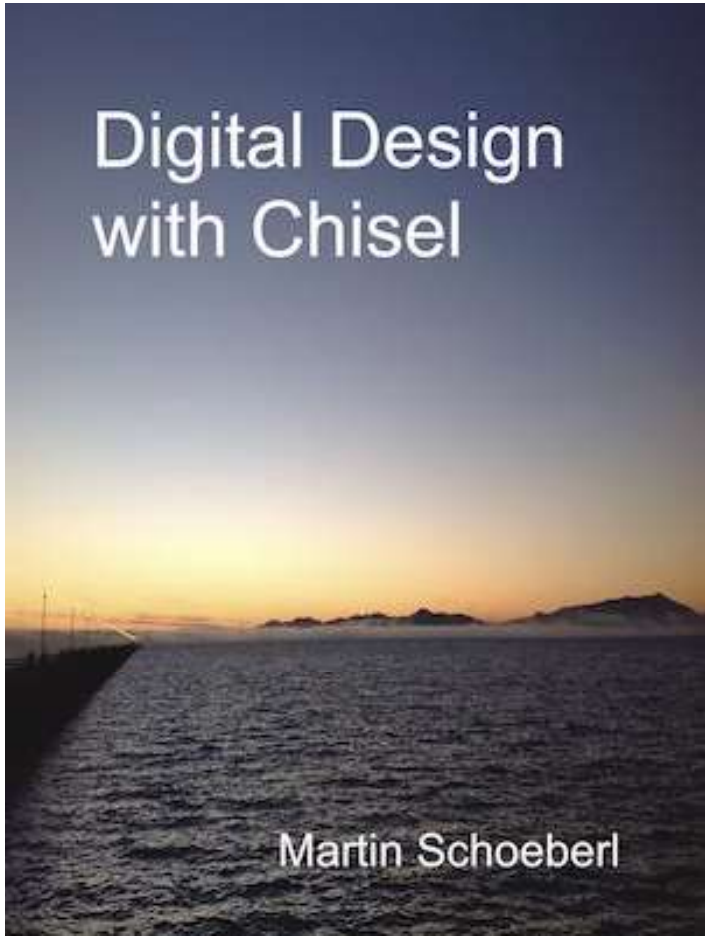
A Small Language

- Chisel is a *small* language
- On purpose
- Not many constructs to remember
- The Chisel Cheatsheet fits on two pages
- The power comes with Scala for circuit generators
- With Scala, Chisel can grow with you

Language Families



The Chisel Book



- Available in open access (as PDF)
  - Optimized for reading on a tablet (size, hyper links)
- Amazon can do the printout

A Taster of Chisel

```
class Hello extends Module {
  val io = IO(new Bundle {
    val led = Output(UInt(1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U;

  val cntReg = RegInit(0.U(32.W))
  val blkReg = RegInit(0.U(1.W))

  cntReg := cntReg + 1.U
  when(cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}
```

Chisel is a Hardware Construction Language

- The code I showed you looks much like Java code
- But it is *not* a program in the usual sense
- It represents a circuit
- The “program” constructs the circuit
- All statements are “executed” in parallel
- Statement order has mostly no meaning

Reminder: We Construct Hardware

- Chisel code looks much like Java code
- But it is *not* a program in the usual sense
- It represents a circuit
- We should be able to *draw* that circuit
- The “program” constructs the circuit
- All statements are “executed” in parallel
- Statement order has mostly no meaning

File Organization in Scala/Chisel

- Designs shall be in `src/main/scala/`
- Tests shall be in `src/test/scala/`

```
project
├── src
│   ├── main
│   │   └── scala
│   │       └── package
│   │           └── sub-package
│   └── test
│       └── scala
│           └── package
├── target
└── generated
```

Coding Style

- Similar to Java
- Use readable, meaningful names
  - E.g., `sum` instead of `y`
- Use `camelCase` for identifiers
- `Modules` (classes) start with uppercase
  - E.g., `ProgramCounter`
- Mark you register with a postfix `Reg`
  - E.g., `countReg`
- Use consistent indentation
  - Chisel style is 2 spaces (blanks)

Common Acronyms

ADC	analog-to-digital converter
ALU	arithmetic and logic unit
ASIC	application-specific integrated circuit
Chisel	constructing hardware in a Scala embedded language
CISC	complex instruction set computer
CRC	cyclic redundancy check
DAC	digital-to-analog converter
DFF	D flip-flop, data flip-flop
DMA	direct memory access
DRAM	dynamic random access memory
FF	flip-flop

Common Acronyms II

FIFO	first-in, first-out
FPGA	field-programmable gate array
HDL	hardware description language
HLS	high-level synthesis
IC	instruction count
IDE	integrated development environment
IO	input/output
ISA	instruction set architecture
JDK	Java development kit
JIT	just-lin-time
JVM	Java virtual machine
LC	logic cell

Common Acronyms III

LRU	least-recently used
MMIO	memory-mapped IO
MUX	multiplexer
OO	object oriented
RISC	reduced instruction set computer
SDRAM	synchronous DRAM
SRAM	static random access memory
TOS	top-of stack
UART	universal asynchronous receiver/transmitter
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuit

Types

Signal Types

- All types in hardware are a collection of bits
- The base type in Chisel is `Bits` (never used!)
- `UInt` represents an unsigned integer
- `SInt` represents a signed integer (in two's complement)

```
Bits(8.W) <-- Never used
UInt(16.W)
SInt(32.W)
```

Number of Bits: n.W

- A collection of bits has a *width*
- The width is the number of bits
- Is written as number followed by `.W`
- Following example shows widths of 8, 16, 32

```
Bits(8.W)
UInt(16.W)
SInt(32.W)
```

Constants

- Constants can represent signed or unsigned numbers
- We use `.U` and `.S` to distinguish

```
0.U  // defines a UInt constant of 0
-3.S // defines a SInt constant of -3
```

- Constants can also be specified with a width

```
3.U(4.W) // An 4-bit constant of 3
```

Hexadecimal and Binary Representation

- We can specify constants with a different base
- May come handy sometimes

```
"hff".U  // hexadecimal representation of 255
"o377".U  // octal representation of 255
"b1111_1111".U  // binary representation of 255
```

Boolean Values

- Type for logical values
- Can be `true` or `false`
- Almost exchangeable with `UInt(1.W)`
- Sometimes a signal, such as `valid`, may be better represented by a Boolean type

```
Bool()
true.B
false.B
```

Operators

Logic Operators

```
val and = a & b // bitwise and
val or = a | b  // bitwise or
val xor = a ^ b // bitwise xor
val not = ~a    // bitwise negation
```

- Note that we do not need to define the width of the values
- Note also that this is *hardware*
- All expressions are evaluated in parallel
- Order does not matter

Comparison Operators

- The usual operations (as in Java or C)
  - Unusual equal and unequal operator symbols
  - To keep the original Sala operators usable for references
- Operands are `UInt` and `SInt`
- Operands can be `Bool` for equal and unequal
- Result is `Bool`

```
>, >=, <, <=

===, =/=
```

Boolean Logical Operators

- Operands and result are `Bool`
- Logical NOT, AND, and OR

```
val notX = !x
val bothTrue = a && b
val orVal = x || y
```

Arithmetic Operators

- Same as in Java or C
- The width of the result is automatically computed
- E.g., the width of the multiplication is the sum of the width of `a` and the width of `b`

```
val add = a + b // addition
val sub = a - b // subtraction
val neg = -a    // negate
val mul = a * b // multiplication
val div = a / b // division
val mod = a % b // modulo operation
```

Chisel Hardware Operators Summary

Operator	Description	Data types
`* / %`	multiplication, division, modulus	UInt, SInt
`+ -`	addition, subtraction	UInt, SInt
`=== =/=`	equal, not equal	UInt, SInt, returns Bool
`> >= < <=`	comparison	UInt, SInt, returns Bool
`<< >>`	shift left, shift right (sign extend on SInt)	UInt, SInt
`~`	NOT	UInt, SInt, Bool
`& | ^`	AND, OR, XOR	UInt, SInt, Bool
`!`	logical NOT	Bool
`&& ||`	logical AND, OR	Bool

Subfields and Concatenation Operators

A single bit can be extracted as follows:

```
val sign = x(31)
```

A subfield can be extracted from end to start position:

```
val lowByte = largeWord(7, 0)
```

Bit fields are concatenated with `Cat`:

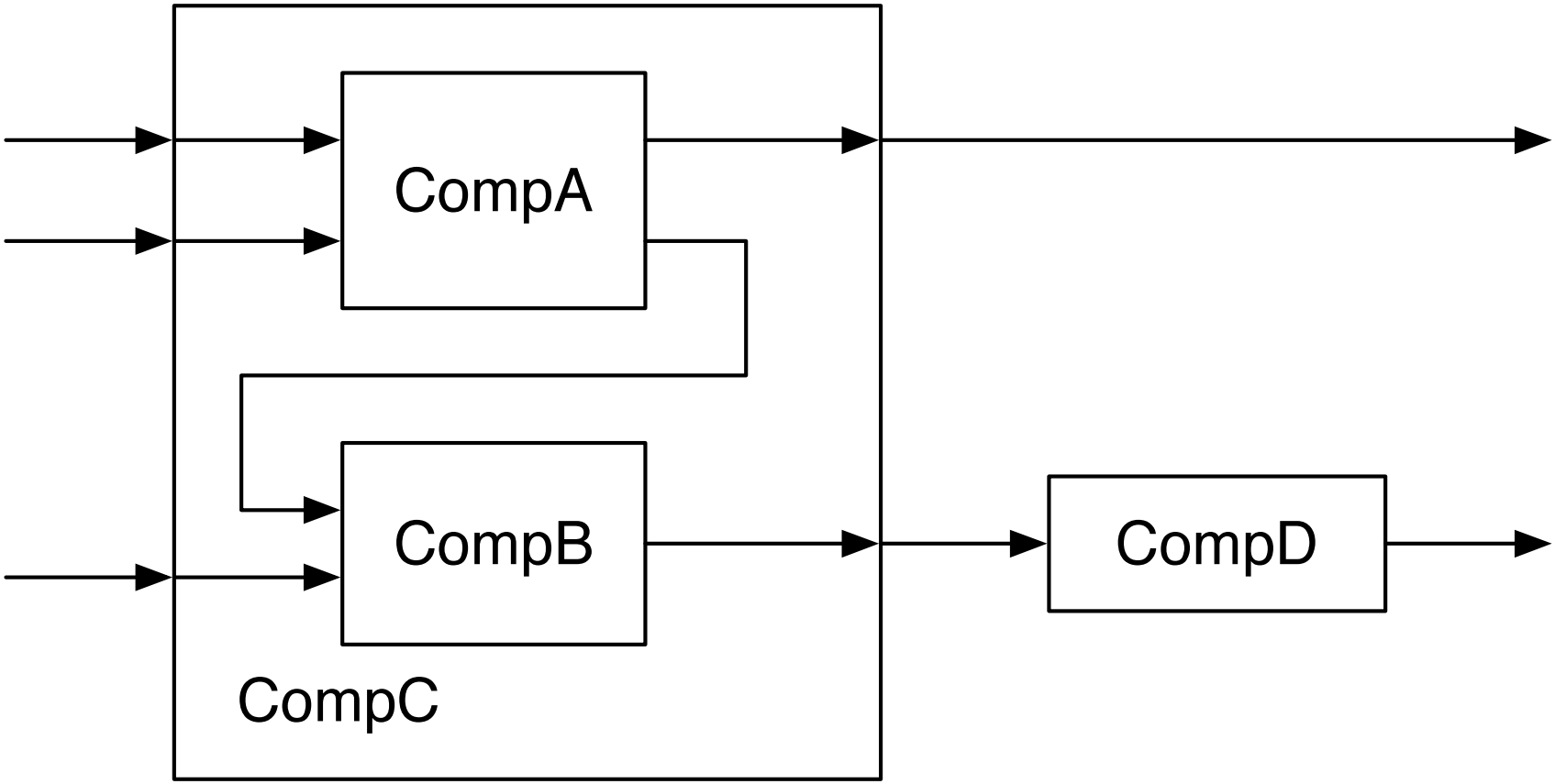
```
val word = Cat(highByte, lowByte)
```

Modules

Components (or Modules)

- Components are building blocks
- Components have input and output ports (= pins)
  - Organized as a `Bundle` assigned to field `io`
- We build circuits as a hierarchy of components
- In Chisel a component is called `Module`
- Components/Modules are used to organize the circuit
  - Similar as using methods in Java

Hierarchy of Components Example



Input/Output Ports

- Ports are bundles with directions
- Ports used to connect modules

```
class AluIO extends Bundle {
  val function = Input(UInt(2.W))
  val inputA = Input(UInt(4.W))
  val inputB = Input(UInt(4.W))
  val result = Output(UInt(4.W))
}
```

An Adder Module

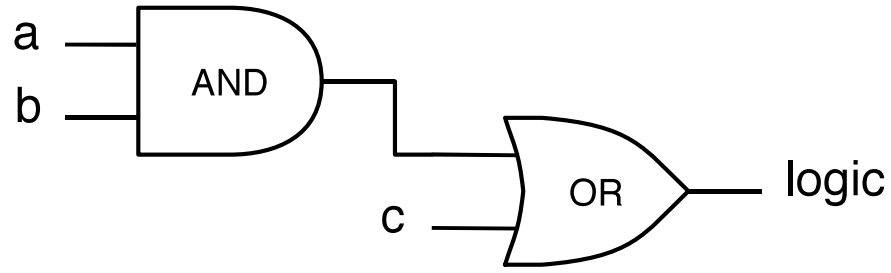
- A `class` that `extends Module`
- Interface (port) is a `Bundle`, wrapped into an `IO()`, and stored in the field `io`
- Circuit description in the constructor

```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val result = Output(UInt(4.W))
  })

  val addVal = io.a + io.b
  io.result := addVal
}
```

Combinational Circuits

Combinational Circuits

- Chisel uses Boolean operators, similar to C or Java
- `&` is the AND operator and `|` is the OR operator
- The following code is the same as the schematics
- `val logic` gives the circuit/expression the name `logic`
- That name can be used in following expressions




```
val logic = (a & b) | c
```

Combinational Circuits

- Simplest is a Boolean expression
- Assigned a name (e)
- This expression can be reused in another expression

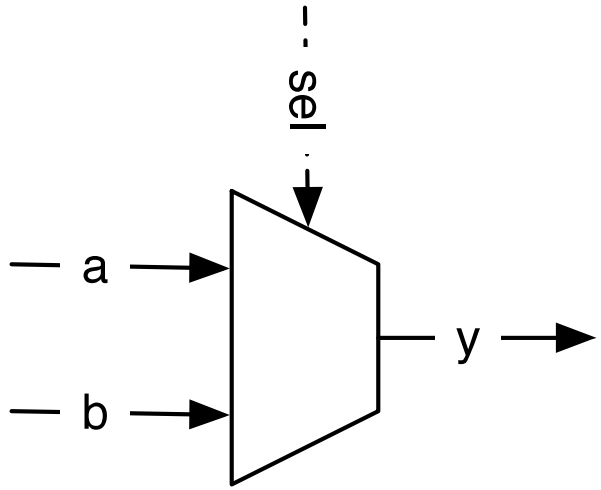
```
val e = (a & b) | c
```

Fixed Expression

- Expression is fixed
- Trying to reassign with = results in an error
- Trying the Chisel conditional update := results in runtime error
- Note the difference between = and :=

```
val e = (a & b) | c

e := c & b
```

A Multiplexer



- A Multiplexer selects between alternatives
- So common that Chisel provides a construct for it
- Selects `a` when `sel` is `true.B` otherwise `b`

```
val result = Mux(sel, a, b)
```

Wires

- A signal (or wire) can be first defined
- And later assigned an expression with `:=`
- Not often needed

```
val w = Wire(Uint(4.W))

w := a & b
```

Combinational Circuit with Conditional Update

- Chisel supports conditional update
- Value first needs to be wrapped into a `Wire`
- Updates with the Chisel update operation `:=`
- With `when` we can express a conditional update
- The resulting circuit is a multiplexer
- The rule is that the last enabled assignment counts
  - Here the order of statements has a meaning

```
val w = Wire(UInt())

w := 0.U
when (cond) {
  w := 3.U
}
```

The “Else” Branch

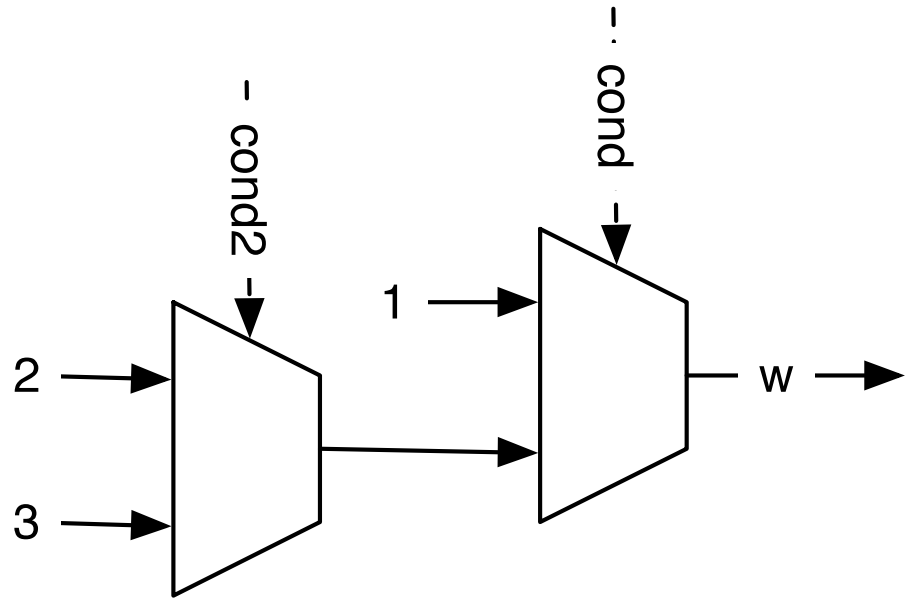
- We can express a form of “else”
- Note the `.` in `.otherwise`

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .otherwise {
  w := 2.U
}
```

A Chain of Conditions

- To test for different conditions
- Select with a priority order
- The first that is true counts
- The hardware is a chain of multiplexers

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .elsewhen (cond2) {
  w := 2.U
} .otherwise {
  w := 3.U
}
```



Default Assignment

- Practical for complex expressions
- Forgetting to assign a value on all conditions
  - Would describe a latch
  - Runtime error in Chisel
- Assign a default value is good practise

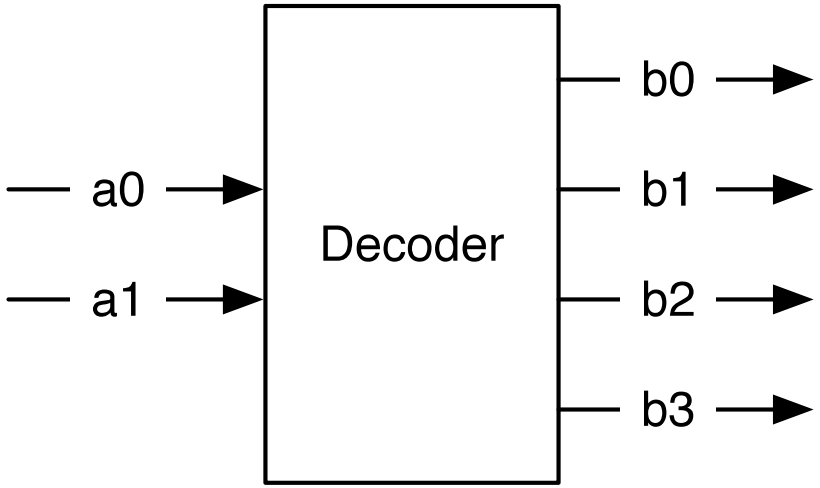
```
val w = WireDefault(0.U)

when (cond) {
  w := 3.U
}
// ... and some more complex conditional
   assignments
```

Logic Can Be Expressed as a Table

- Sometimes more convenient
- Still combinational logic (gates)
- Is converted to Boolean expressions
- Let the synthesize tool do the conversion!
- We use the `switch` statement

```
result := 0.U

switch(sel) {
  is (0.U) { result := 1.U}
  is (1.U) { result := 2.U}
  is (2.U) { result := 4.U}
  is (3.U) { result := 8.U}
}
```

A Decoder



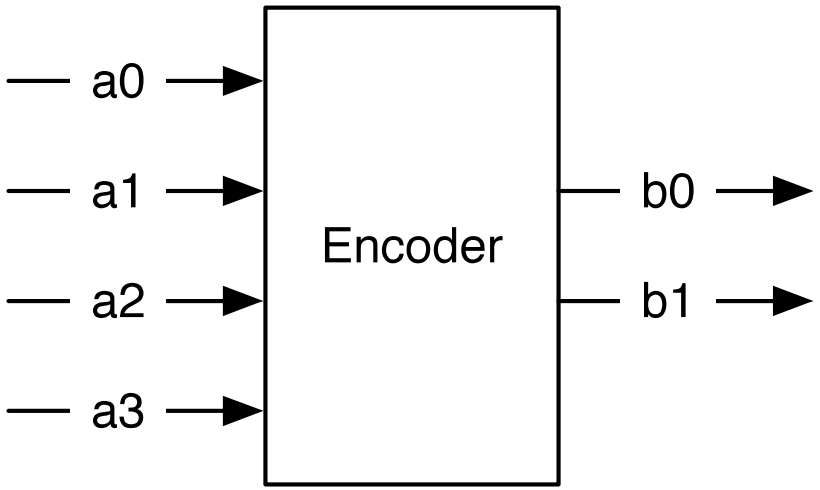
a	b
00	0001
01	0010
10	0100
11	1000

Decoder in Chisel

- Binary strings are a clearer representation

```
switch (sel) {
  is ("b00".U) { result := "b0001".U}
  is ("b01".U) { result := "b0010".U}
  is ("b10".U) { result := "b0100".U}
  is ("b11".U) { result := "b1000".U}
}
```

An Encoder



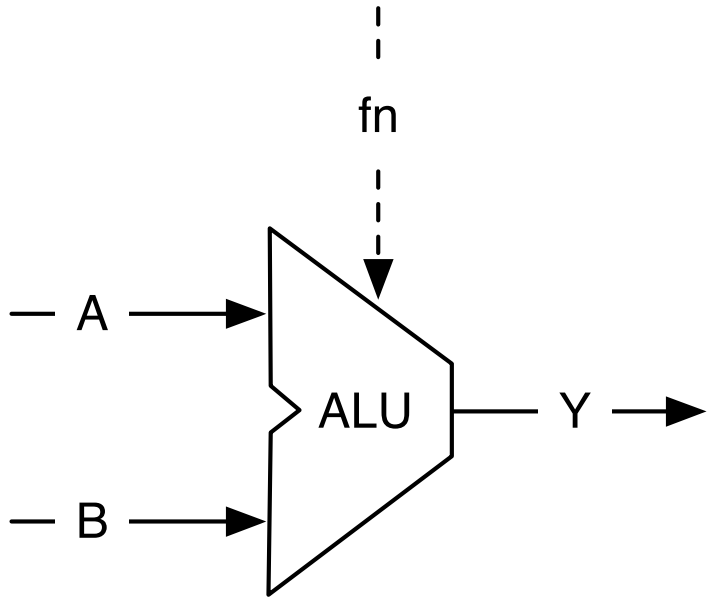
a	b
0001	00
0010	01
0100	10
1000	11
????	??

Encoder in Chisel

- We cannot describe a function with undefined outputs
- We use a default assignment of "b00"

```
b := "b00".U
switch (a) {
  is ("b0001".U) { b := "b00".U}
  is ("b0010".U) { b := "b01".U}
  is ("b0100".U) { b := "b10".U}
  is ("b1000".U) { b := "b11".U}
}
```

Arithmetic Logic Unit



- Also called ALU
- A central component of a microprocessor
- Two inputs, one function select, and an output

Arithmetic Logic Unit in Chisel (Full Module)

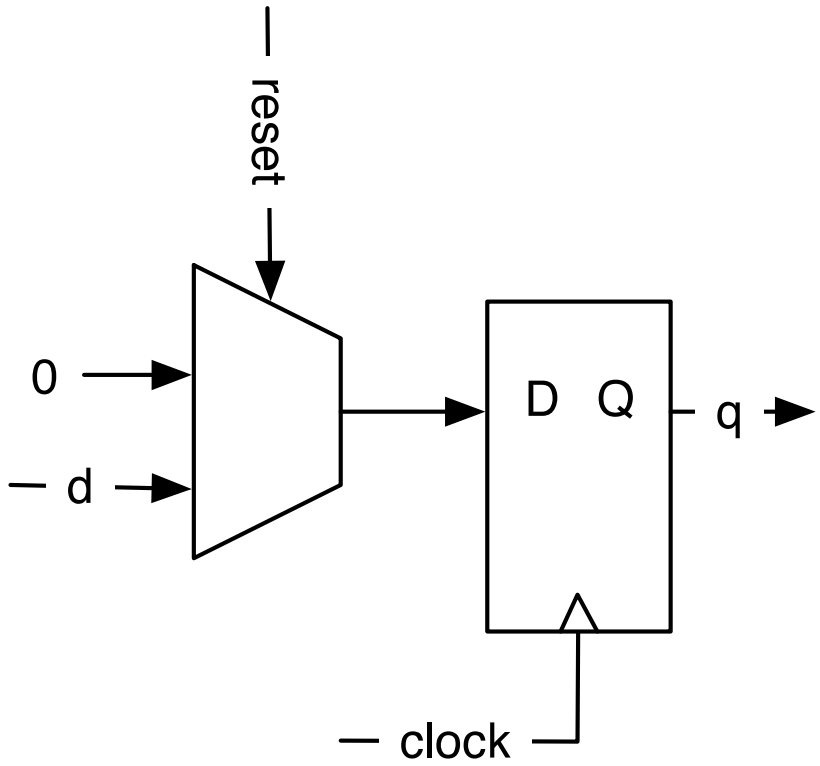
```
class Alu extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(16.W))
    val b = Input(UInt(16.W))
    val fn = Input(UInt(2.W))
    val y = Output(UInt(16.W))
  })

  // some default value is needed
  io.y := 0.U

  // The ALU selection
  switch(io.fn) {
    is(0.U) { io.y := io.a + io.b }
    is(1.U) { io.y := io.a - io.b }
    is(2.U) { io.y := io.a | io.b }
    is(3.U) { io.y := io.a & io.b }
  }
}
```

Sequential Circuits (Using Registers)

Register

- A register is a collection of flip-flops
- Updated on the rising edge of the clock
- May be set to a value on reset
- Clock and reset are implicitly connected to the register
- A register can be any Chisel type that can be represented as a collection of bits

A Register with Reset



A Register with Reset

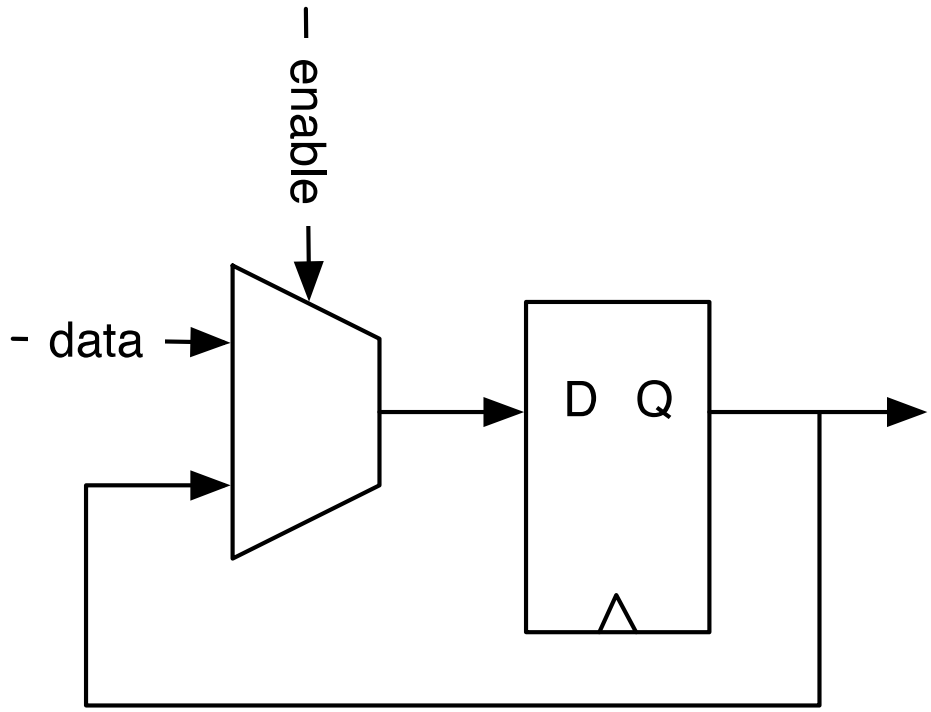
Following code defines an 8-bit register, initialized with 0 at reset:

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

An input is connected to the register with the `:=` update operator and the output of the register can be used just with the name in an expression:

```
reg := d
val q = reg
```

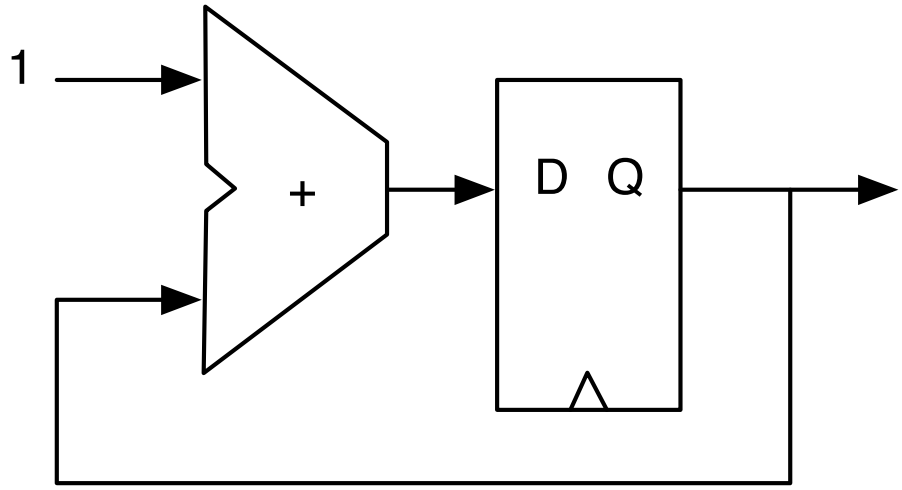
Register with Enable



- Only when `enable` true is a value is stored

```
val enableReg = Reg(UInt(4.W))

when (enable) {
  enableReg := inVal
}
```

Combine a Register with an Adder



- Is a free running counter
- 0, 1, ... 14, 15, 0, 1, ...

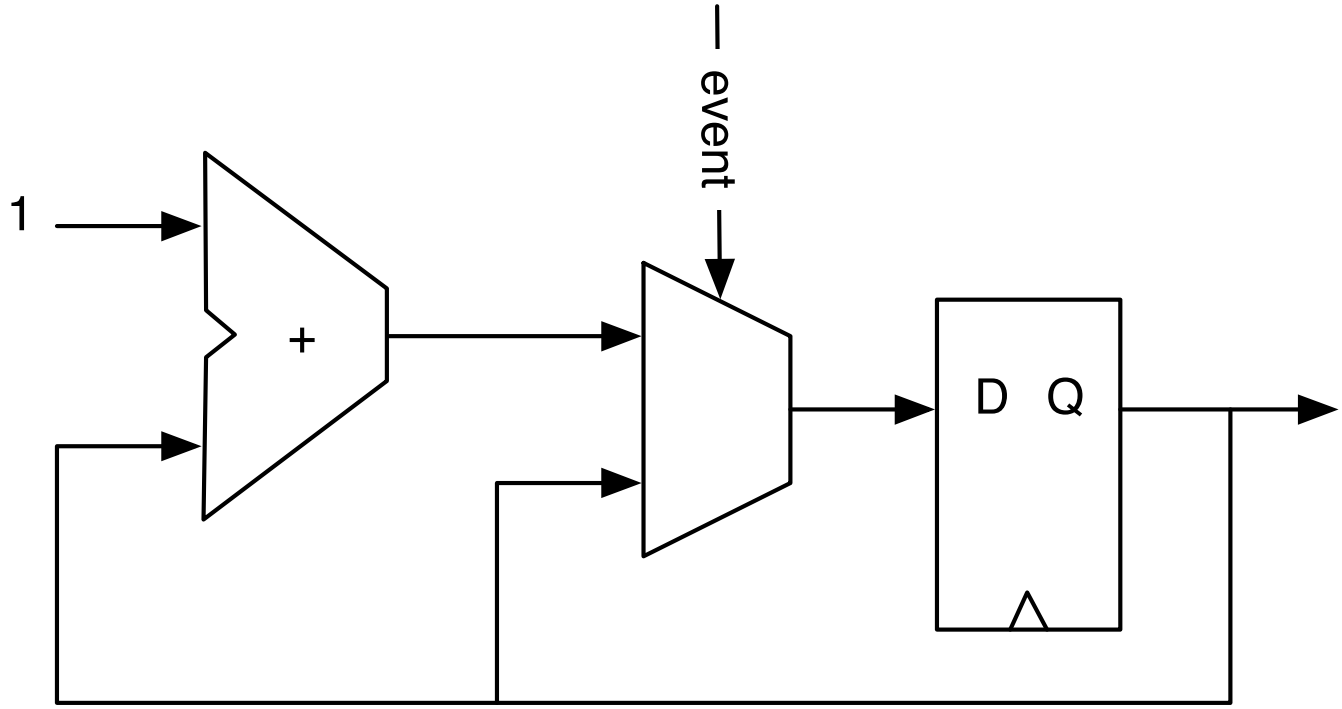
```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
```

A Counter with a Mux

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

cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```

- This counter counts from 0 to 9
- And starts from 0 again after reaching 9
  - Starting from 0 is common in computer engineering
- A counter is the hardware version of a *for loop*
- Often needed

Counting Events



- Is the schematics and the code the same?

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

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
}
```

Collections

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

```
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 multplexer
- 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)
```

Bundles

- A `Bundle` to groups signals
- Can be different types
- Defined by a class that extends `Bundle`
- List the fields as `val`s within the block

```
class Channel() extends Bundle {
  val data = UInt(32.W)
  val valid = Bool()
}
```

Using a Bundle

- Create it with `new`
- Wrap it into a `Wire`
- Field access with *dot* notation

```
val ch = Wire(new Channel())
ch.data := 123.U
ch.valid := true.B

val b = ch.valid
```

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))
}
```

Connecting Modules

An Adder Module

- A `class` that `extends Module`
- Interface (port) is a `Bundle`, wrapped into an `IO()`, and stored in the field `io`
- Circuit description in the constructor

```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val result = Output(UInt(4.W))
  })

  val addVal = io.a + io.b
  io.result := addVal
}
```

Connections

- Simple connections just with assignments, e.g.,

```
adder.io.a := ina
adder.io.b := inb
```

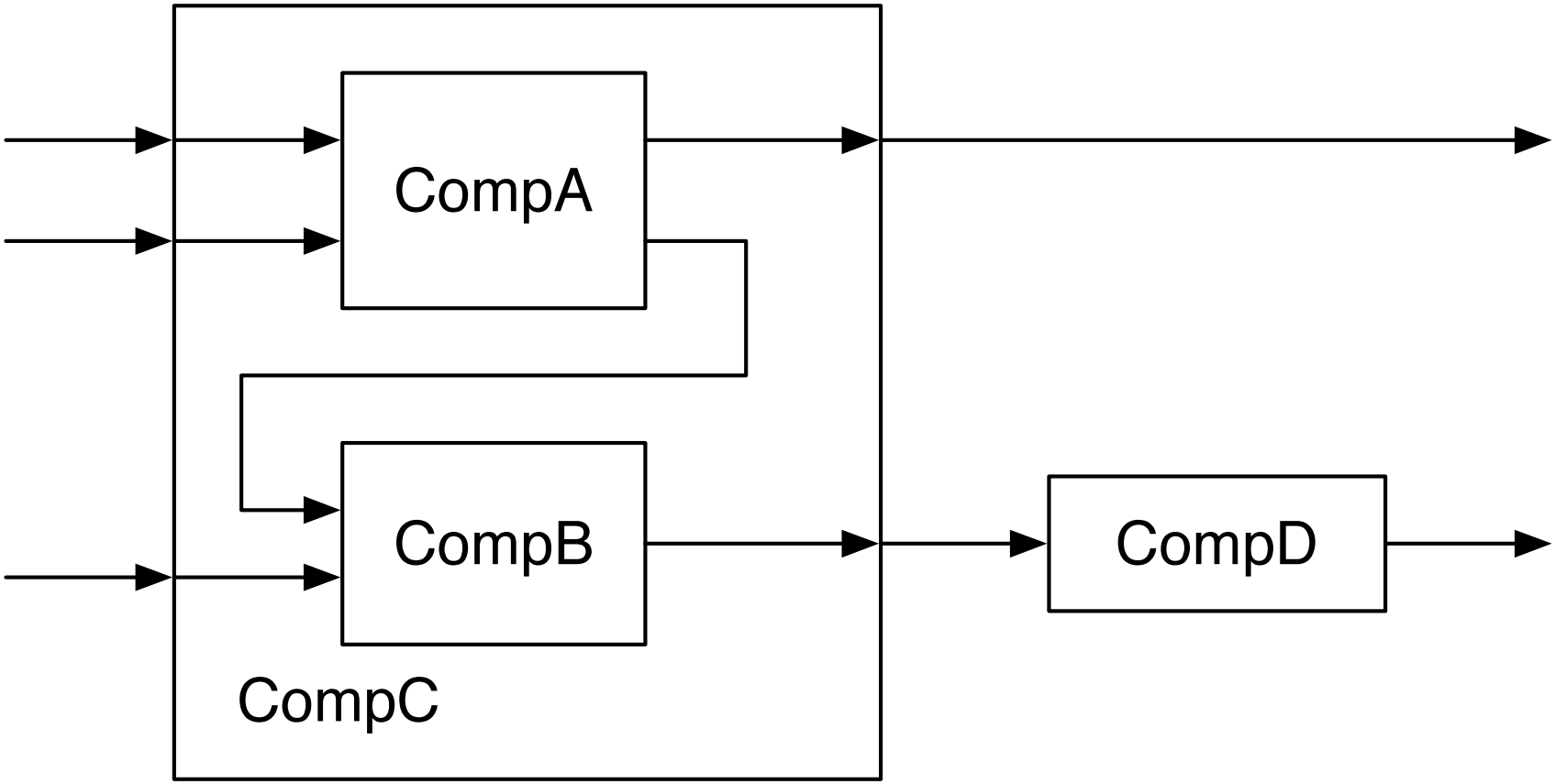
- Note the dot access to the field `io` and then the IO field

Module Use

- Create with `new` and wrap into a `Module()`
- Interface port via the `io` field
- Note the assignment operator `:=` on `io` fields

```
val adder = Module(new Adder())
adder.io.a := ina
adder.io.b := inb
val result = adder.io.result
```

Hierarchy of Components Example



Components CompA and CompB

```
class CompA extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val x = Output(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  // function of A
}

class CompB extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of B
}
```

Component CompC

```
class CompC extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_x = Output(UInt(8.W))
    val out_y = Output(UInt(8.W))
  })

  // create components A and B
  val compA = Module(new CompA())
  val compB = Module(new CompB())

  // connect A
  compA.io.a := io.in_a
  compA.io.b := io.in_b
  io.out_x := compA.io.x
  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.in_c
```

Testing and Debugging

Testing and Debugging

- Nobody writes perfect code ;-)
- We need a method to improve the code
- In Java we can simply print the result:
  - `println("42");`
- What can we do in hardware?
  - Describe the whole circuit and hope it works?
  - We can switch an LED on or off
- We need some tools for debugging
- Writing testers in Chisel

Testing with Chisel

- 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 chisel3.iotesters._
```

Using peek, poke, and expect

```
// Set input values
poke(dut.io.a, 3)
poke(dut.io.b, 4)
// Execute one iteration
step(1)
// Print the result
val res = peek(dut.io.result)
println(res)

// Or compare against expected value
expect(dut.io.result, 7)
```

A Chisel Tester

- Extends class `PeekPokeTester`
- Has the device-under test (DUT) as parameter
- Testing code can use all features of Scala

```
class CounterTester(dut: Counter) extends
    PeekPokeTester(dut) {

  // Here comes the Chisel/Scala code
  // for the testing
}
```

Example DUT

- A device-under test (DUT)

```
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 Simple Tester

- Just using `println` for manual inspection

```
class TesterSimple(dut: DeviceUnderTest)
    extends PeekPokeTester(dut) {

  poke(dut.io.a, 0.U)
  poke(dut.io.b, 1.U)
  step(1)
  println("Result is: " +
      peek(dut.io.out).toString)
  poke(dut.io.a, 3.U)
  poke(dut.io.b, 2.U)
  step(1)
  println("Result is: " +
      peek(dut.io.out).toString)
}
```

The Main Program for the Test

- Extend an App and invoke the `iotesters` driver
- With the DUT and the tester

```
object TesterSimple extends App {
  chisel3.iotesters.Driver(() => new
      DeviceUnderTest()) { c =>
    new TesterSimple(c)
  }
}
```

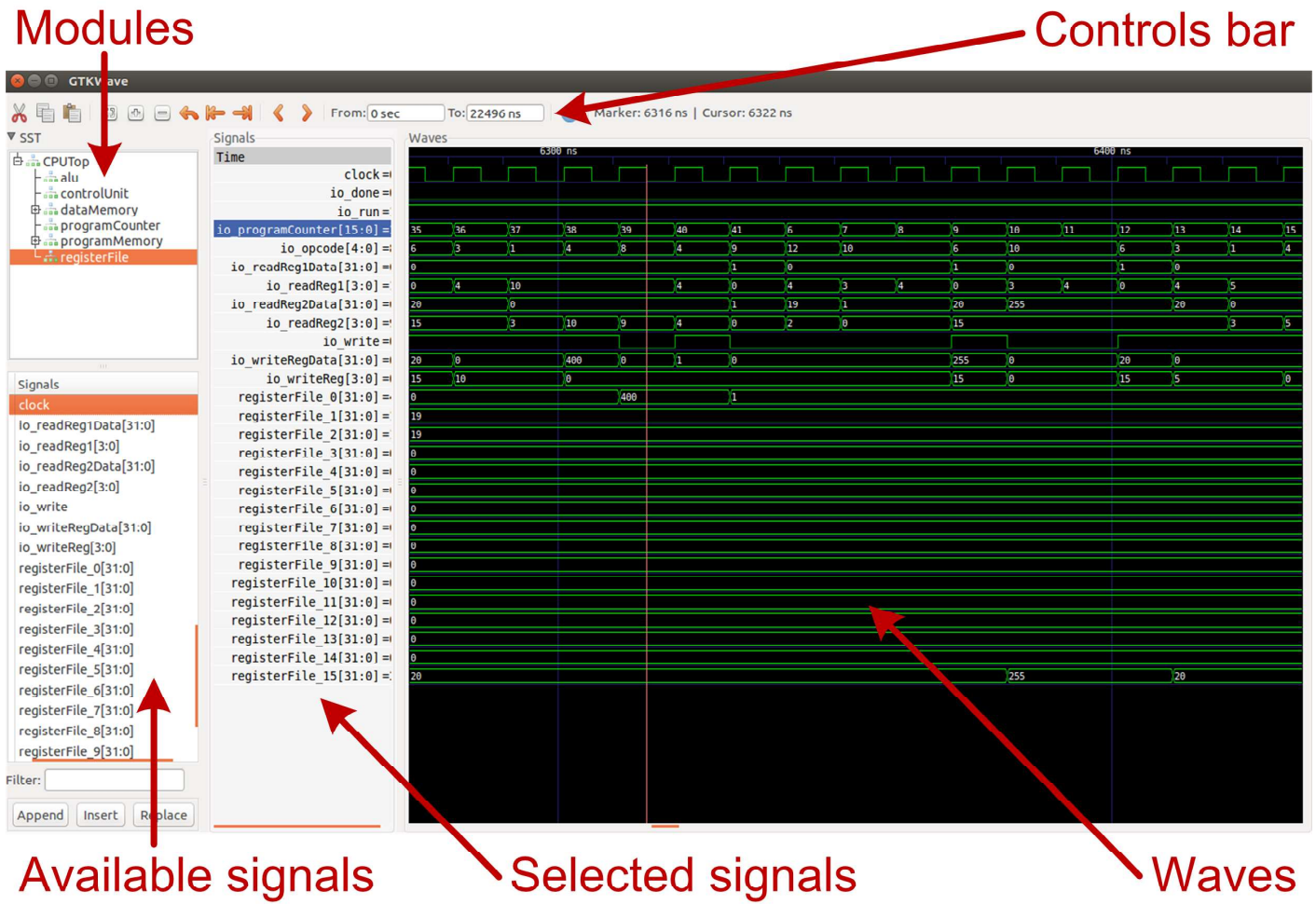
A Real Tester

- Poke values and `expect` some output

```
class Tester(dut: DeviceUnderTest) extends
    PeekPokeTester(dut) {

  poke(dut.io.a, 3.U)
  poke(dut.io.b, 1.U)
  step(1)
  expect(dut.io.out, 1)
  poke(dut.io.a, 2.U)
  poke(dut.io.b, 0.U)
  step(1)
  expect(dut.io.out, 0)
}
```

Waveforms (The Most Useful Tool)

- Simulation stores a `.vcd` file
- Open it with GTKWave and see all your signals in parallel



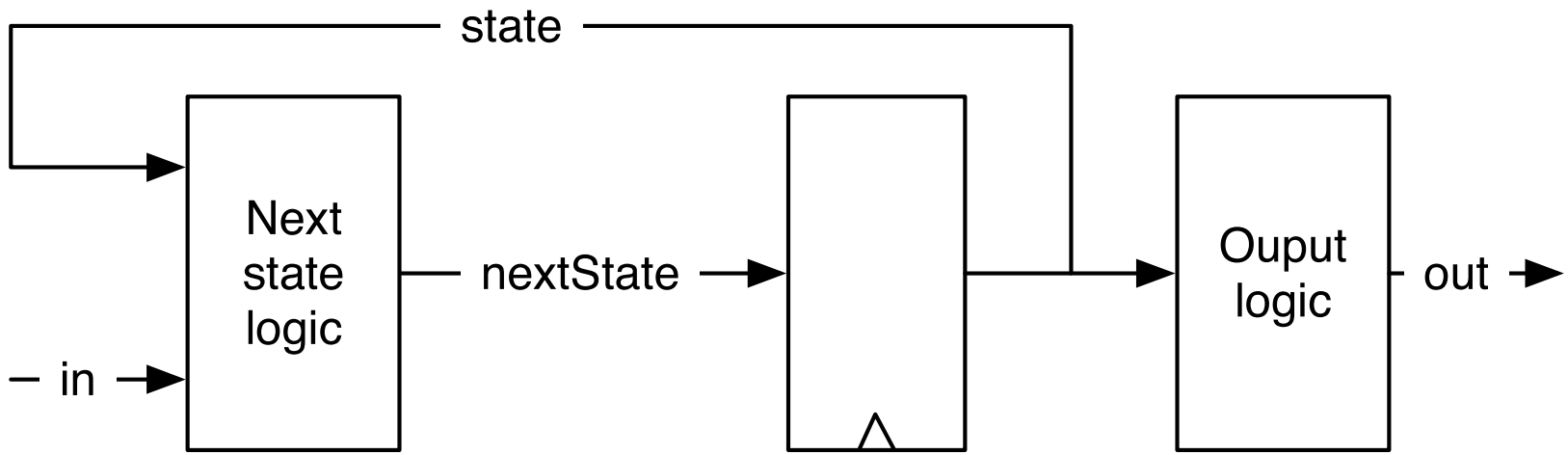
Finite State Machines (with Datapath)

Finite-State Machine (FSM)

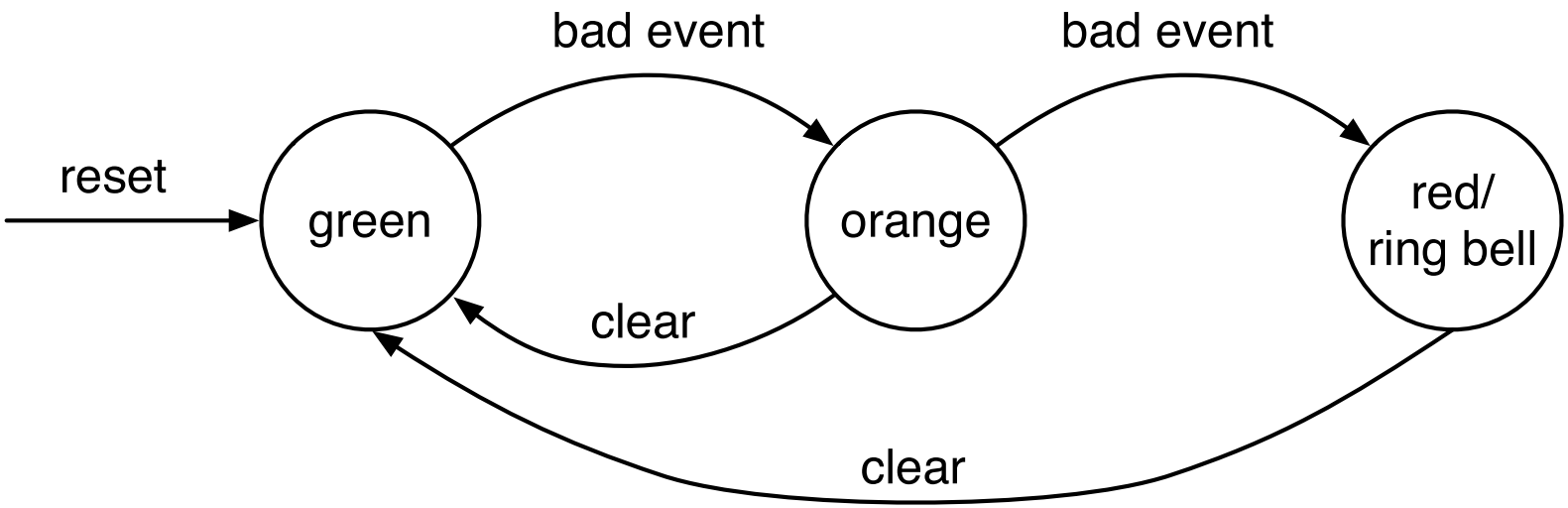
- Has a register that contains the state
- Has a function to computer the next state
  - Depending on current state and input
- Has an output depending on the state
  - And maybe on the input as well
- Every synchronous circuit can be considered a finite state machine
- However, sometimes the state space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

	Input			
State	Bad event	Clear	Next state	Ring bell
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

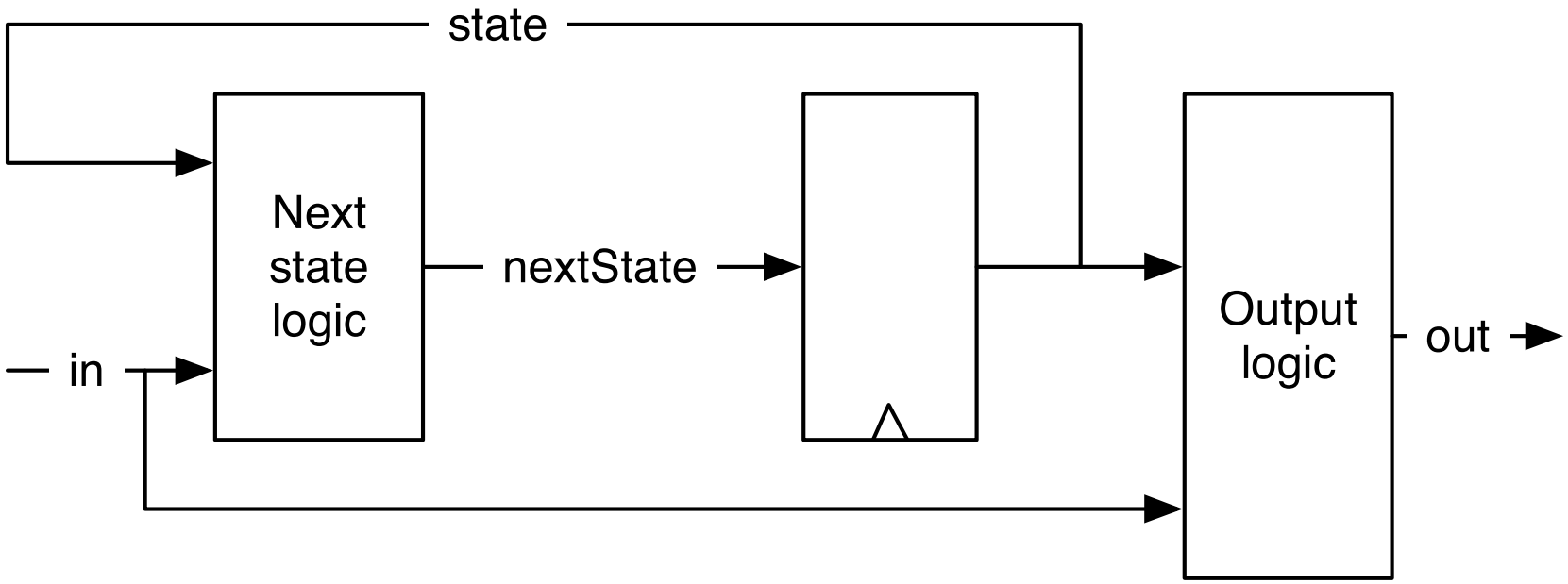
```
io.ringBell := stateReg === red
```

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

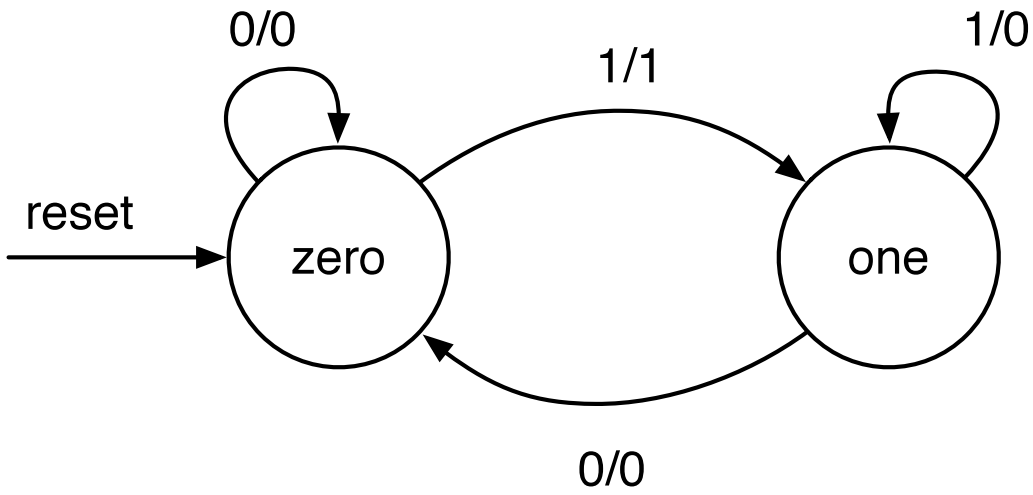
A Mealy FSM

- 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

- The code is in the book as it is too long for slides

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, ...

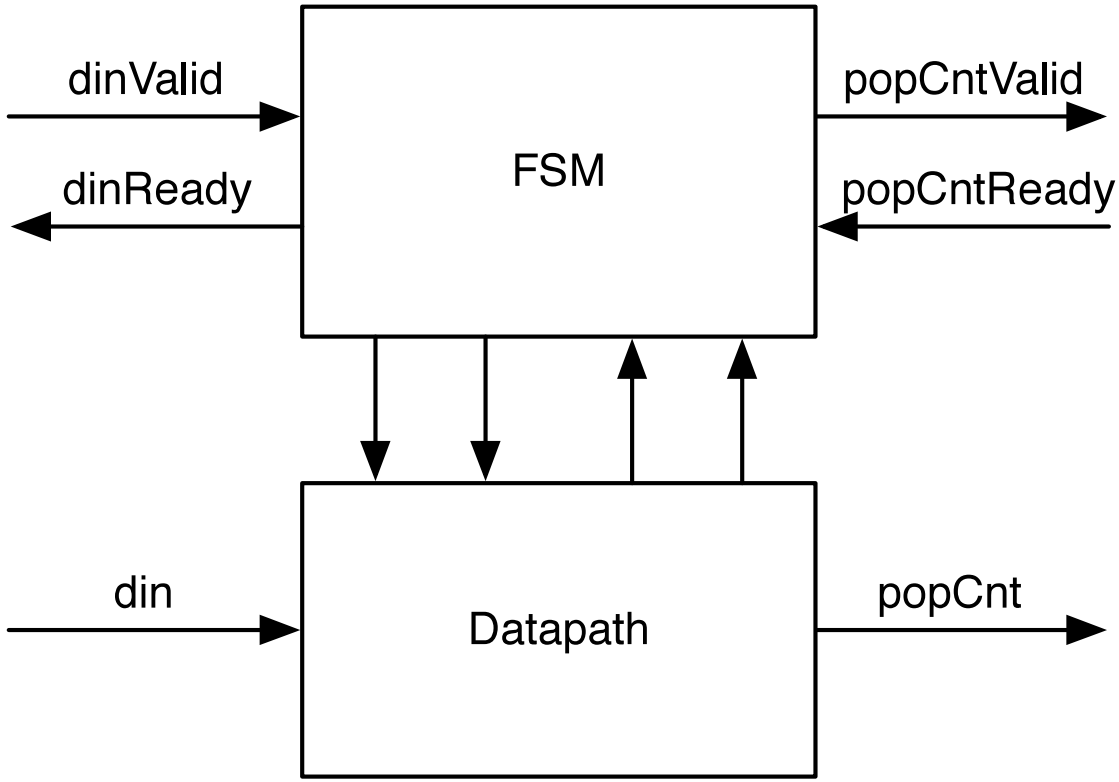
FSM-Datapath Interaction

- 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

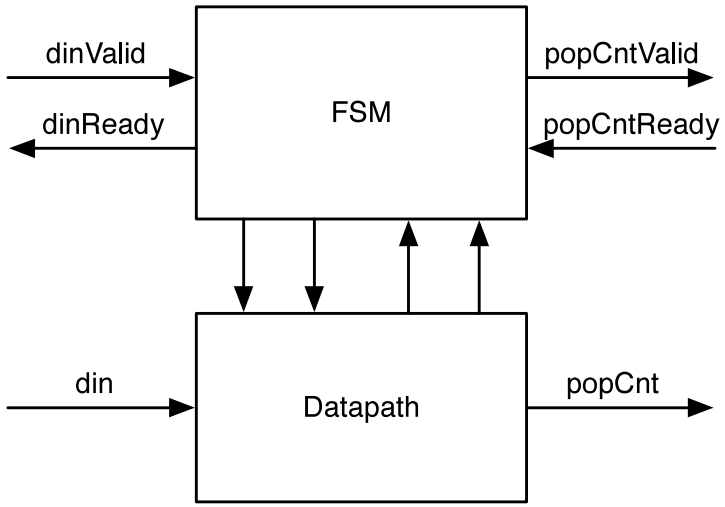
- An FSMD that computes the popcount
- Also called the Hamming weight
- Compute the number of ‘1’s in a word
- Input is the data word
- Output is the count
- Code available at PopCount.scala

Popcount Block Diagram



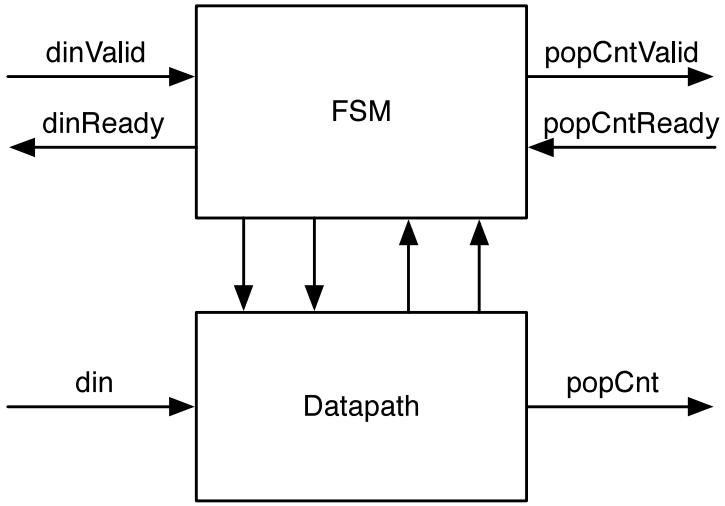
Popcount Connection

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

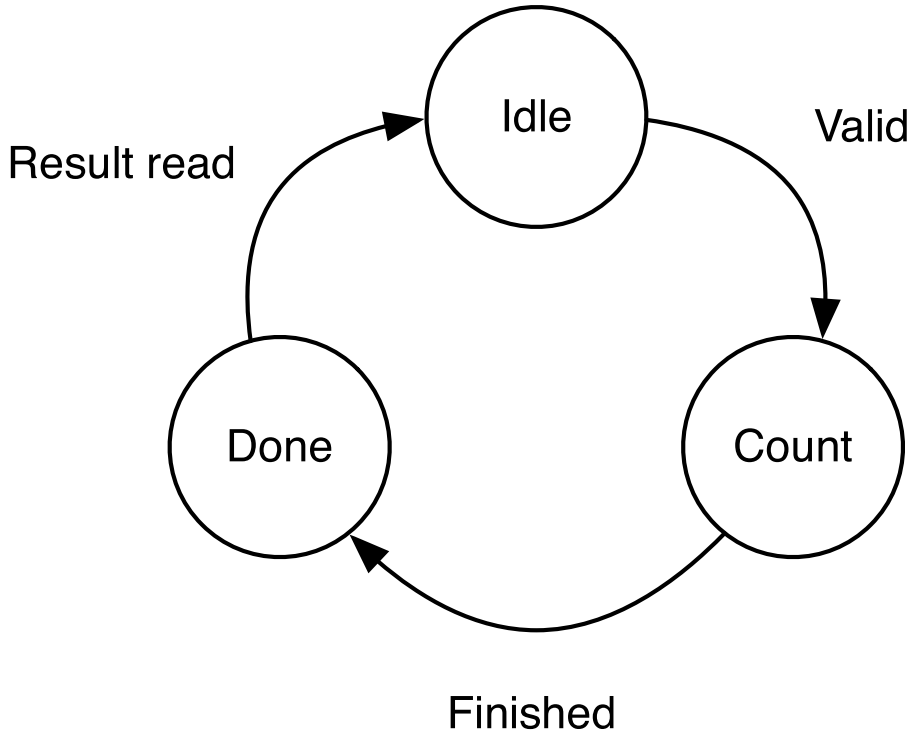


Popcount Handshake

- 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

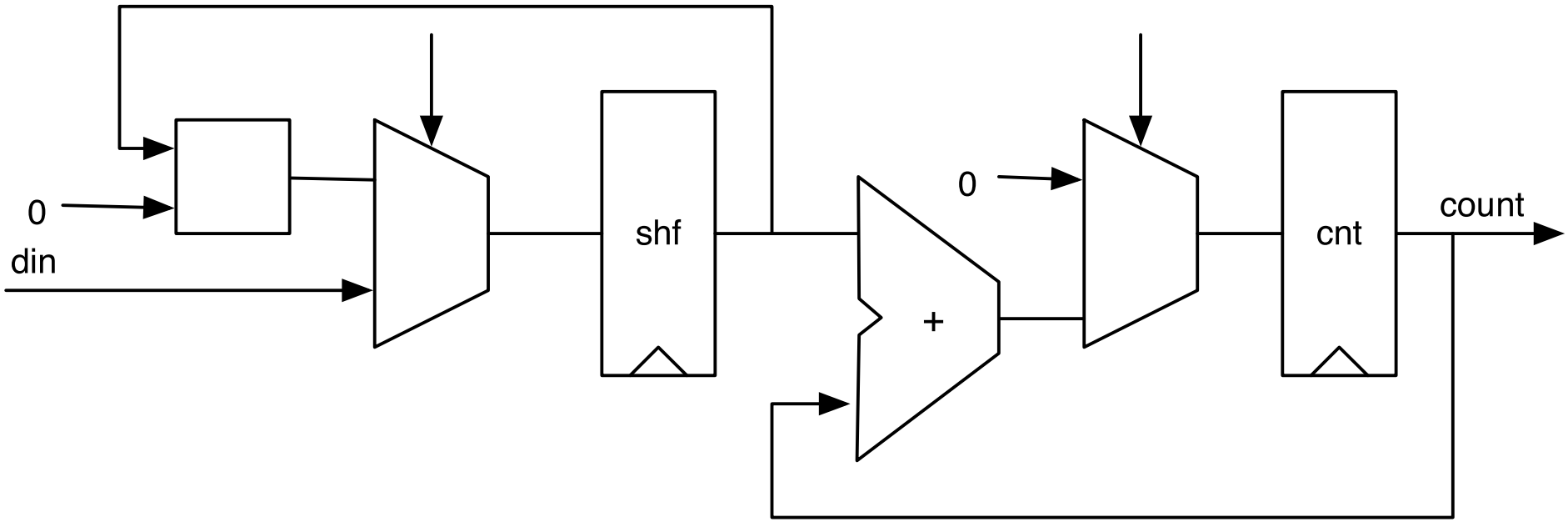


The FSM



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

The Datapath



Explore the Code

- In PopCount.scala