CS250
VLSI Systems Design
Lecture 2: Chisel Introduction

Spring 2016

John Wawrzynek
with
Chris Yarp (GSI)

HDL History

- Verilog originated at Automated Integrated Design Systems (renamed Gateway) in 1985. Acquired by Cadence in 1989.
- Invented as simulation language. Synthesis was an afterthought. Many of the basic techniques for synthesis were developed at Berkeley in the 80's and applied commercially in the 90's.
- Around the same time as the origin of Verilog, the US Department of Defense developed VHDL (A double acronym! VSIC (Very High-Speed Integrated Circuit) HDL). Because it was in the public domain it began to grow in popularity.
- Afraid of losing market share, Cadence opened Verilog to the public in 1990.
- An IEEE working group was established in 1993, and ratified IEEE Standard 1394 (Verilog) in 1995.
- Verilog is the language of choice of Silicon Valley companies, initially because of high-quality tool support and its similarity to C-language syntax.
- VHDL is still popular within the government, in Europe and Japan, and some Universities.
- Most major CAD frameworks now support both.
- Latest Verilog version is "System Verilog" .
- Other alternatives these days:
  - Bluespec (MIT spin-out) models digital systems using "guarded atomic actions"
  - C-to-gates Compilers (ex: Cadence C-to-s, Vivado HLS)

Problems with Verilog

- Designed as a simulation language. "Discrete Event Semantics"
  - Many constructs don't synthesize: ex: deassign, timing constructs
  - Others lead to mysterious results: for-loops
  - Difficult to understand synthesis implications of procedural assignment (always blocks), and blocking versus non-blocking assignments
  - Your favorite complaint here!
  - In common use, most users ignore much of the language and stick to a very strict "style". Large companies post use rules and run lint style checkers. Nonetheless

- The real power of a textual representation of circuits is the ability to write circuit "compilers". Verilog has very weak "meta-programming" support". Simple parameter expressions, generate loops and case.
- Various hacks around this over the years, ex: embedded TCL scripting.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
     parameter SIZE = 8;
     output [SIZE-1:0] bin;
     input  [SIZE-1:0] gray;

     genvar i;

     generate for (i=0; i<SIZE; i=i+1) begin:bit
        assign bin[i] = ^gray[SIZE-1:i];
     end endgenerate
  endmodule
```

Chisel

Constructing Hardware In a Scala Embedded Language

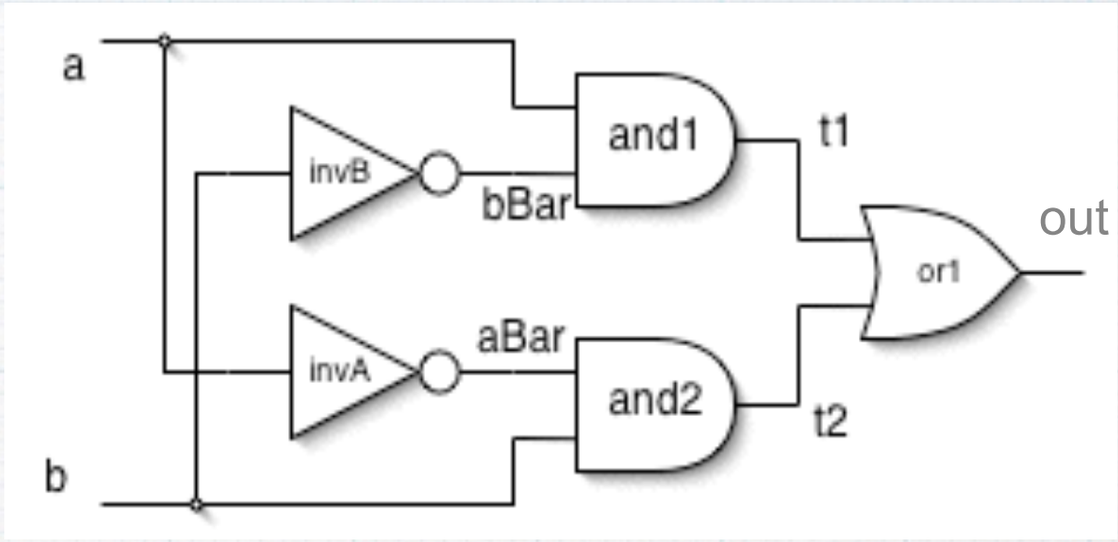
- Experimental attempt at a fresh start to address these issues.
  - Clean simple set of design construction primitives, just what is needed for RTL design
  - Powerful "metaprogramming" model for building circuit generators
- Why embedded?
  - Avoid the hassle of writing and maintaining a new programming language (most of the work would go into the non-hardware specific parts of the language anyway).
- Why Scala?
  - Brings together the best of many others: Java JVM, functional programming, OO programming, strong typing, type inference.
- Still very new. Bugs will show up. Your feedback is needed.
- In class, brief presentation of basics. Ask questions.
- Tutorial/manual and other documents available online: chisel.berkeley.edu
- Note: Chisel is not High-level Synthesis. Much closer to Verilog/VHDL than C-to-gates.

Outline

- Brief Introduction to Chisel
- Literal Constructors
- Bundles, Port Constructors, Vecs
- Components and Circuit Hierarchy
- More on Multiplexors
- Registers
- Conditional Update Rules
- FSMs
- More on Interface Bundles, and Bulk Connections
- Running

Simple Combinational Logic Example

```
// simple logic expression
(a & ~b) | (~a & b)
```

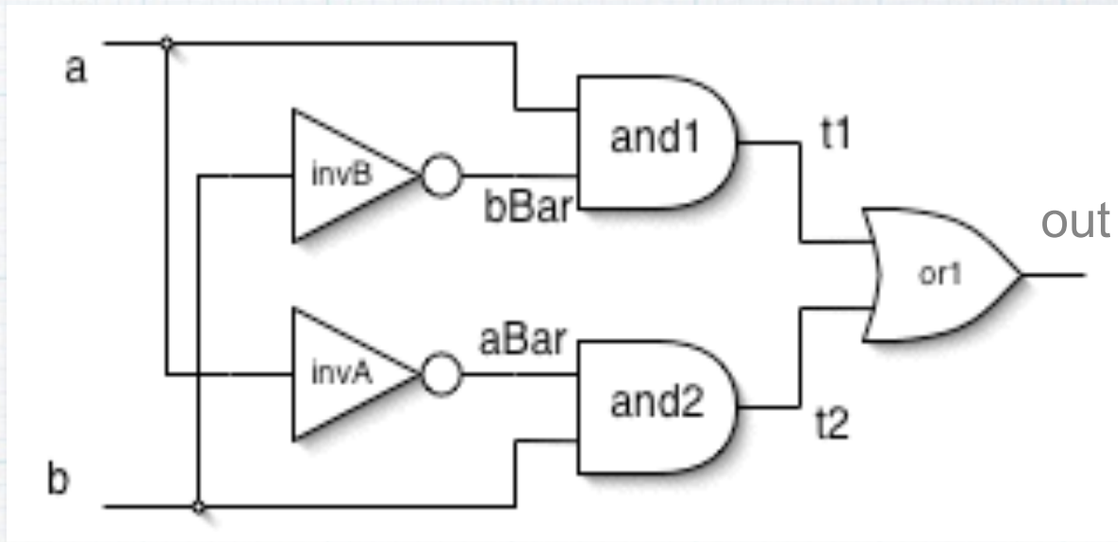


- Notes:
  - The associated logic circuits are not "executed". They are active always (like continuous assignment in Verilog).
  - Unlike Verilog, no built-in logic gates. Expressions instead.
  - The "variables", `a` and `b`, are "named wires", and were given names here because they are inputs to the circuit. Other wires don't need names.
  - Here we assumed that the inputs, and therefore all generated wires, are one bit wide, but the same expression would work for wider wires. The logic operators used here are "bitwise". There are corresponding operations for booleans.
    - Chisel includes a powerful wire width inference mechanism.

Simple Combinational Logic Example

- In the previous example because the wires `a` and `b`, are named, each can be used in several places. Similarly we could name the circuit output:

```
// simple logic expression

val out =(a & ~b) | (~a & b)
```



- The keyword `val` comes from Scala. It is a way to declare a program variable that can only be assigned once - a constant.
- This way `out` can be generated at one place in the circuit and then "fanned-out" to other places where `out` appears.

```
// fan-out

val z =(a & out) | (out & b)
```

- Another reason to name a wire is to help in debugging.

Functional Abstraction

- Naming wires and using fanout gives us a way to reuse an output in several places in the generated circuit. Function abstraction gives us a way to reuse a **circuit description**:

```
// simple logic function

def XOR (a: Bits, b: Bits) = (a & ~b) | (~a & b)
```

  - Here the function inputs and output are assigned the type `Bits`. More on types soon.

- Now, wherever we use the `XOR` function, we get a copy of the associated logic. Think of the function as a "constructor".

```
// Constructing multiple copies

val z =(x & XOR(x,y)) | (XOR(x,y) & y)
```

  - Functions wrapping up simple logic are light-weight. This results in hierarchy in your code, but no hierarchy in the Chisel output.
  - We'll see later that **Chisel Modules** are used for building hierarchy in the resulting circuit.

Datatypes in Chisel

- **Chisel datatypes are used to specify the type of values held in state elements or flowing on wires.**
- Hardware circuits ultimately operate on vectors of binary digits, but more abstract representations for values allow clearer specifications and help the tools generate more optimal circuits.
- The basic types in Chisel are:

`Bits`	Raw collection of bits
`SInt`	Signed integer number
`UInt`	Unsigned integer number
`Bool`	Boolean

- All signed numbers represented as 2's complement.
- Chisel supports several higher-order types: Bundles and Vecs.

Type Inference

- **Although it is useful to keep track of the types of your wires, because of Scala type inference, it is not always necessary to declare the type.**
- For instance in our earlier example:

```
// simple logic expression
val out =(a & ~b) | (~a & b)
```

the type of out was inferred from the types of `a` and `b` and the operators.

- If you want to make sure, or if there is not enough information around for the inference engine, you can always specify the type explicitly:

```
// simple logic expression
val out: Bits =(a & ~b) | (~a & b)
```

- Also, as we shall see, explicit type declaration is necessary in some situations.

Bundles

- **Chisel Bundles represent collections of wires with named fields.**
- Similar to "struct" in C. In chisel, Bundles are defined as a class (similar to in C++ and Java):

```
class FIFOInput extends Bundle {
  val rdy = Bool(OUTPUT)          // Indicates if FIFO has space
  val data = Bits(INPUT, 32)      // The value to be enqueued
  val enq = Bool(INPUT)           // Assert to enqueue data
}
```

- Chisel has class methods for Bundle (i.e., automatic connection creation) therefore user created bundles need to "extend" class Bundle. (More later)
- Each field is given a name and defined with a constructor of the proper type and with parameters specifying width and direction.
- Instances of `FIFOInput` can now be made: `val jonsIO = new FIFOInput;`
  - Bundle definitions can be nested and built into hierarchies,
  - And are used to define the interface of "modules" …
  - Bundle "flip" operator is used to create the "opposite" Bundle (wrt to direction)

Literals

- Literals are values specified directly in your source code.
- Chisel defines type specific constructors for specifying literals.

```
Bits("ha")      // hexadecimal 4-bit literal of type Bits
Bits("o12")     // octal 4-bit literal of type Bits
Bits("b1010")   // binary 4-bit literal of type Bits
SInt(5)         // signed decimal 4-bit literal of type Fix
SInt(-8)        // negative decimal 4-bit literal of type Fix
UInt(5)         // unsigned decimal 3-bit literal of type UFix
Bool(true)      // literals for type Bool, from Scala boolean literals
Bool(false)
```

- By default Chisel will size your literal to the minimum necessary width.
- Alternatively, you can specify a width value as a second argument:

```
Bits("ha", 8)    // hexadecimal 8-bit literal of type Bits, 0-extended
SInt(-5, 32)     // 32-bit decimal literal of type Fix, sign-extended
SInt(-5, width = 32) // handy if lots of parameters
```

- Error reported if specified width value is less than needed.

Builtin Operators

- Chisel defines a set of hardware operators for the builtin types.

Bool Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	1
`x && y`	Logical AND	1
`x || y`	Logical OR	1

Bits Operators:

Chisel	Explanation	Width
`x(n)`	Extract bit, 0 is LSB	1
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y)`
`x >> y`	Dynamic right shift	`w(x) - minVal(y)`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate x, n times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If c, then x; else y	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	1
`x != y`	Inequality	1
`andR(x)`	AND-reduce	1
`orR(x)`	OR-reduce	1
`xorR(x)`	XOR-reduce	1

UInt, SInt Operators: (bitwdths given for `UInts`)

Chisel	Explanation	Width
`x + y`	Addition	`max(w(x), w(y))`
`x - y`	Subtraction	`max(w(x), w(y))`
`x * y`	Multiplication	`w(x) + w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y) - 1)`
`x > y`	Greater than	1
`x >= y`	Greater than or equal	1
`x < y`	Less than	1
`x <= y`	Less than or equal	1
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

Bit-width Inference

- **A nice feature of the Chisel compiler is that it will automatically size the width of wires.**
- The bit-width of ports (of modules) and registers must be specified, but otherwise widths are inferred with the application of the following rules:

`z = x + y`	wz = max(wx, wy)
`z = x - y`	wz = max(wx, wy)
`z = x <bitwise-op> y`	wz = max(wx, wy)
`z = Mux(c, x, y)`	wz = max(wx, wy)
`z = w * y`	wz = wx + wy
`z = x << n`	wz = wx + maxNum(n)
`z = x >> n`	wz = wx - minNum(n)
`z = Cat(x, y)`	wz = wx + wy
`z = Fill(n, x)`	wz = wx * maxNum(n)

Bundles and Vecs

- **Bundle and Vec are classes for aggregates of other types.**
- The Bundle class similar to "struct" in C, collection with named fields:

```
class MyFloat extends Bundle {
  val sign = Bool()
  val exponent = Bits(width = 8)
  val significant = Bits(width = 23)
}
val x = new MyFloat()
Val xs = x.sign
```

- The Vec class is an indexable array of same type objects:

```
val myVec = Vec(5) { SInt(width = 23) }  // Vec of 5 23-bit signed integers.
val third = myVec(3) // Name one of the 23-bit signed integers
```

  - Note: Vec can contain collections of wires, registers, or bundles.
  - Vec and Bundle inherit from class, **Data**. Every object that ultimately inherits from Data can be represented as a bit vector in a hardware design.
  - Nesting:

```
class BigBundle extends Bundle {
  val myVec = Vec(5) { SInt(width = 23) } // Vector of 5 23-bit signed integers.
  val flag = Bool()
  val f = new MyFloat() // Previously defined bundle.
}
```

Ports

- A port is any `Data` object with directions assigned to its members.
- Port constructors allow a direction to be added at construction time:

```
class FIFOInput extends Bundle {
  val rdy =  Bool(OUTPUT)
  val data = Bits(width = 32, OUTPUT)
  val enq =  Bool(INPUT)
}
```

- The direction of an object can also be assigned at instantiation time (although is

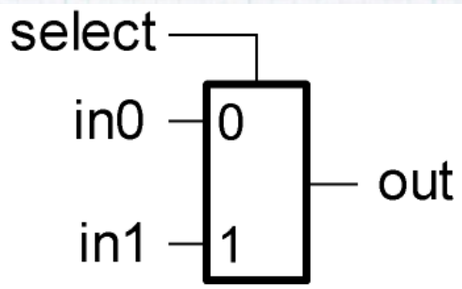
```
class ScaleIO extends Bundle {
  val in = new MyFloat().asInput
  val scale = new MyFloat().asInput
  val out = new MyFloat().asOutput
}
```

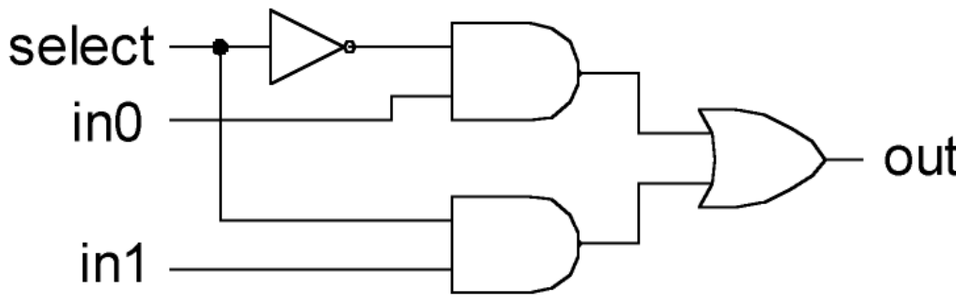
- The methods asInput and asOutput force all components of the data object to the requested direction.
- Other methods exist for "flipping" direction, etc.

Modules

- **Modules are used to define hierarchy in the generated circuit.**
- Similar to modules in Verilog.
- Each defines a port interface, wires together subcircuits.
- Module definitions are class definitions that extend the Chisel Module class.

```
class Mux2 extends Module {
  val io = new Bundle{
    val select = Bits(width=1, dir=INPUT)
    val in0 = Bits(width=1, dir=INPUT)
    val in1 = Bits(width=1, dir=INPUT)
    val out = Bits(width=1, dir=OUTPUT)
  }
  io.out := (io.select & io.in1) |
            (~io.select & io.in0)
}
```



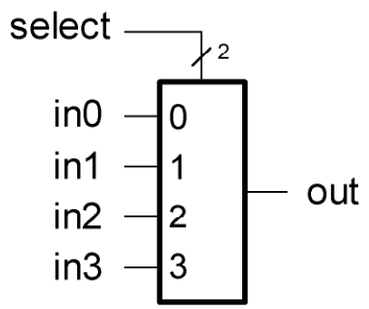


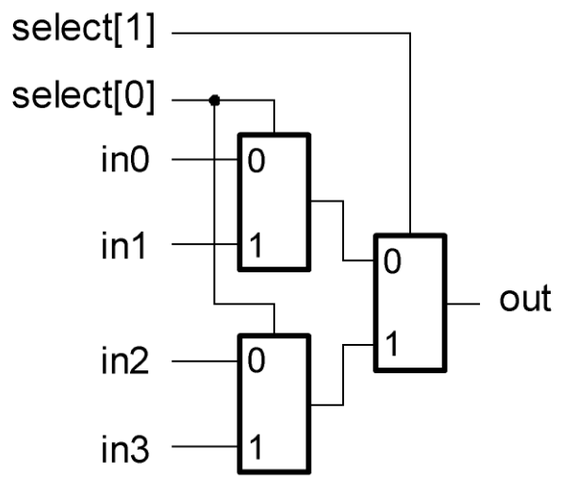
- The Module slot `io` is used to hold the interface definition, of type Bundle. `io` is assigned a Bundle that defines its ports.
- In this example,
  - `io` is assigned to an **anonymous** Bundle,
  - ":=" assignment operator, in Chisel wires the input of LHS to the output of circuit on the RHS

Component Instantiation

- Modules are used to define hierarchy in the generated circuit.

```
class Mux4 extends Module {
  val io = new Bundle {
    val in0    = Bits(width=1, dir=INPUT)
    val in1    = Bits(width=1, dir=INPUT)
    val in2    = Bits(width=1, dir=INPUT)
    val in3    = Bits(width=1, dir=INPUT)
    val select = Bits(width=2, dir=INPUT)
    val out    = Bits(width=1, dir=OUTPUT)
  }
  val m0 = new Mux2();
  m0.io.select := io.select(0); m0.io.in0 := io.in0; m0.io.in1 := io.in1;

  val m1 = new Mux2();
  m1.io.select := io.select(0); m1.io.in0 := io.in2; m1.io.in1 := io.in3;

  val m3 = new Mux2()
  m3.io.select := io.select(1);
  m3.io.in0 := m0.io.out; m3.io.in1 := m1.io.out

  io.out := m3.io.out
}
```





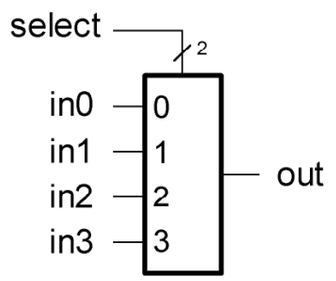
Component Functional Abstraction

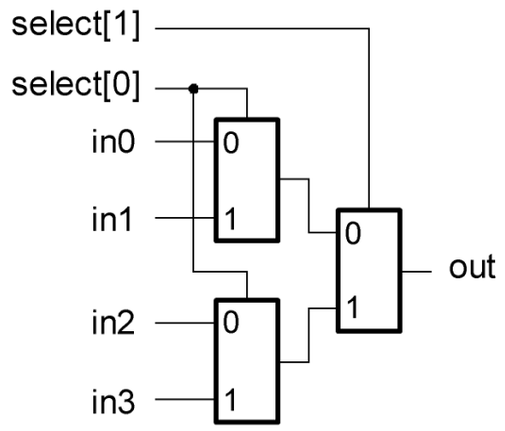
- Functional constructors for Modules can simplify your code.

```
object Mux2 {
  def apply (select: Bits, in0: Bits, in1: Bits) = {
    val m = new Mux2()
    m.io.in0 := in0
    m.io.in1 := in1
    m.io.select := select
    m.io.out                          // return the output
  }
}
```

- object Mux2 creates a Scala singleton object on the Mux2 component class.
- apply defines a method for creation of a Mux2 instance

```
class Mux4 extends Component {
  val io = new Bundle {
    val in0    = Bits(width=1, dir=INPUT)
    val in1    = Bits(width=1, dir=INPUT)
    val in2    = Bits(width=1, dir=INPUT)
    val in3    = Bits(width=1, dir=INPUT)
    val select = Bits(width=2, dir=INPUT)
    val out = Bits(1, OUTPUT)
  };
  io.out := Mux2(io.select(1),
                 Mux2(io.select(0), io.in0, io.in1),
                 Mux2(io.select(0), io.in2, io.in3))
}
```





More on Multiplexors

- **Chisel defines a constructor for n-way multiplexors**

```
MuxLookup(index, default,
          Array(key1->value1, key2->value2,..., keyN->valueN))
```

- The index to key match is implemented using the "===" operator.
- Therefore MuxLookup would work for any type for which === is defined.
- "===" is defined on bundles and vecs, as well as the primitive Chisel types.
- Users might can override "===" for their own bundles.

- **MuxCase generalizes this by having each key be an arbitrary condition**

```
MuxCase(default, Array(c1 -> a, c2 -> b, ...))
```

- where the overall expression returns the value corresponding to the first condition evaluating to true.

Registers

- Simplest form of state element supported by Chisel is a positive-edge-triggered register. Is instantiated functionally as:

```
Reg((a & ~b) | (~a & b))
```

- This circuit has an output that is a copy of the input signal delayed by one clock cycle.
- Note, we do not have to specify the type of Reg as it will be automatically inferred from its input when instantiated in this way.
- In Chisel, clock and reset are global signals that are implicitly included where needed
- Example use. Rising-edge detector that takes a boolean signal in and outputs true when the current value is true and the previous value is false:

```
def risingedge(x: Bool) = x && !Reg(x)
```

The Counter Example

- Constructor for an up-counter that counts up to a maximum value, `max`, then wraps around back to zero (i.e., modulo `max+1`):

```
def wraparound(n: UInt, max: UInt) =
  Mux(n > max, UInt(0), n)

def counter(max: UInt) = {
  val y = Reg(resetVal = UInt(0, max.getWidth))
  y := wraparound(y + UInt(1), max)
  y
}
```

- Constructor for a circuit to output a pulse every `n` cycles:

```
// Produce pulse every n cycles.
def pulse(n: UInt) = counter(n - UInt(1)) === UInt(0)
```

- "Toggle flip-flop" - toggles internal state when `ce` is true:

```
// Flip internal state when input true.
def toggle(ce: Bool) = {
  val x = Reg(resetVal = Bool(false))
  x := Mux(ce, !x, x)
  x
}
```

```
def squareWave(period: UInt) = toggle(pulse(period))
```

Conditional Updates

- Instead of wiring register inputs to combinational logic blocks, it is often useful to specify when updates to the registers will occur and to specify these updates spread across several separate statements (think FSMs).

```
val r = Reg() { UInt(width = 16) }
  when (c === 0) {
    r := r + UInt(1)
  }
```

- register r is updated on the next rising-clock-edge iff c is zero.
- The argument to when is a predicate circuit expression that returns a Bool.

- When a value is assigned in multiple when blocks, the last when block that is true takes precedence

```
r := SInt(3); s := SInt(3)
when (c1) { r := SInt(1); s := SInt(1) }
when (c2) { r := SInt(2) }
```

- Leads to:

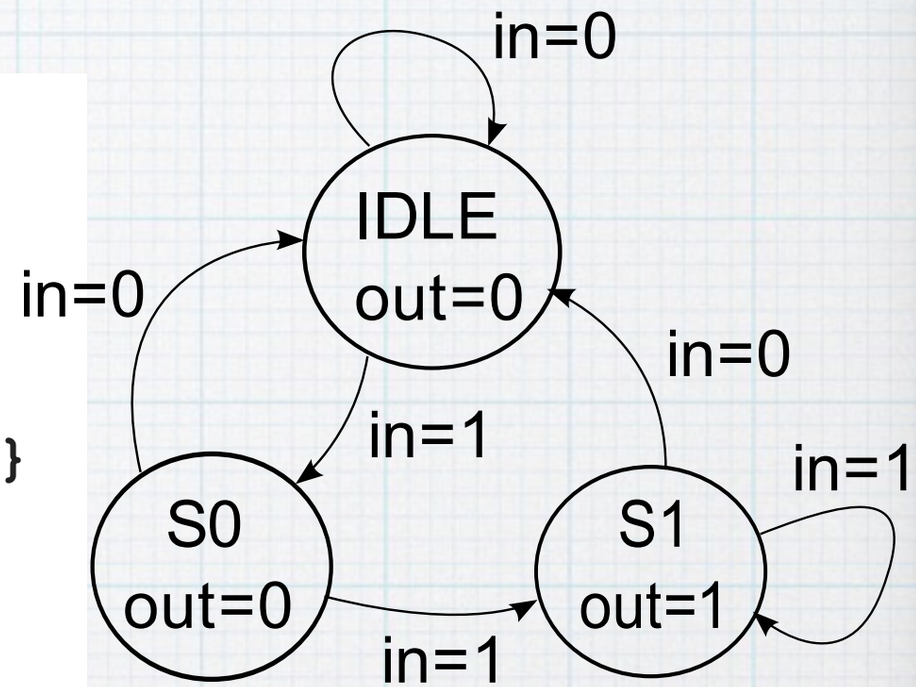
c1	c2	r	s
0	0	3	3
0	1	2	3
1	0	1	1
1	1	2	1

- See tutorial for more examples, and variations on this them.

Finite State Machine Specification (1)

- When blocks help in FSM specification:

```
class MyFSM extends Module {
  val io = new Bundle {
    val in  = Bool(dir = INPUT)
    val out = Bool(dir = OUTPUT)
  }
  val IDLE :: S0 :: S1 :: Nil = Enum(3){UInt()}
  val state = Reg(resetVal = IDLE)
  when (state === IDLE) {
    when (io.in) { state := S0 }
  }
  when (state === S0) {
    when (io.in) { state := S1 }
    .otherwise { state := IDLE }
  }
  when (state === S1) {
    .unless (io.in) { state := IDLE }
  }
  io.out := state === S1
}
```

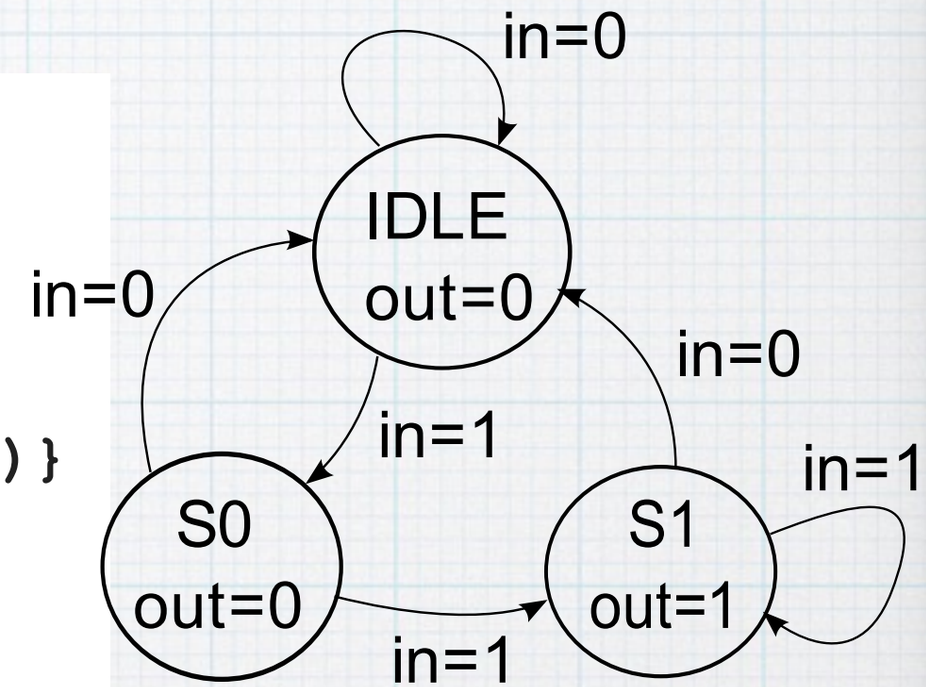


- Enum(3) generates three Uint lits, used here to represent states values.
- See tutorial for more complex FSM example.

Finite State Machine Specification (2)

- Switch helps in FSM specification:

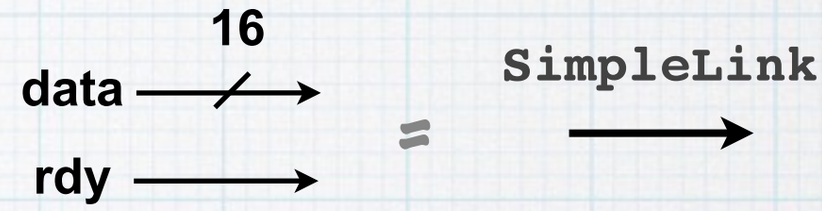
```
class MyFSM extends Component {
  val io = new Bundle {
    val in = Bool(dir = INPUT)
    val out = Bool(dir = OUTPUT)
  }
  val IDLE :: S0 :: S1 :: Nil = Enum(3) {UInt()}
  val state = Reg(resetVal = IDLE)
  switch (state) {
    is (IDLE) {
      when (io.in) { state := S0 }
    }
    is (S0) {
      when (io.in) { state := S1 }
      .otherwise { state := IDLE }
    }
    is (S1) {
    .unless (io.in) { state := IDLE }
    }
  }
  io.out := state === S1
}
```



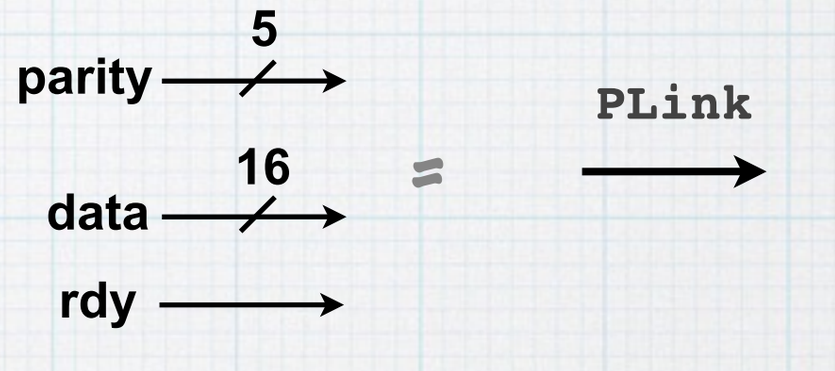
Interfaces and Bulk Connections (1)

- Bundles help with interface definitions

```
class SimpleLink extends Bundle {
  val data = Bits(width=16, dir=OUTPUT)
  val rdy  = Bool(dir=OUTPUT);
}
```

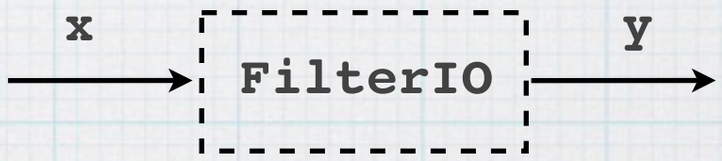



```
// Bundle Inheritance
class PLink extends SimpleLink {
  val parity = Bits(width=5, dir=OUTPUT)
}
```



- `PLink` extends `SimpleLink` by adding parity bits.

```
// Super Bundle through nesting
class FilterIO extends Bundle {
  val x = new PLink().flip
  val y = new PLink()
}
```



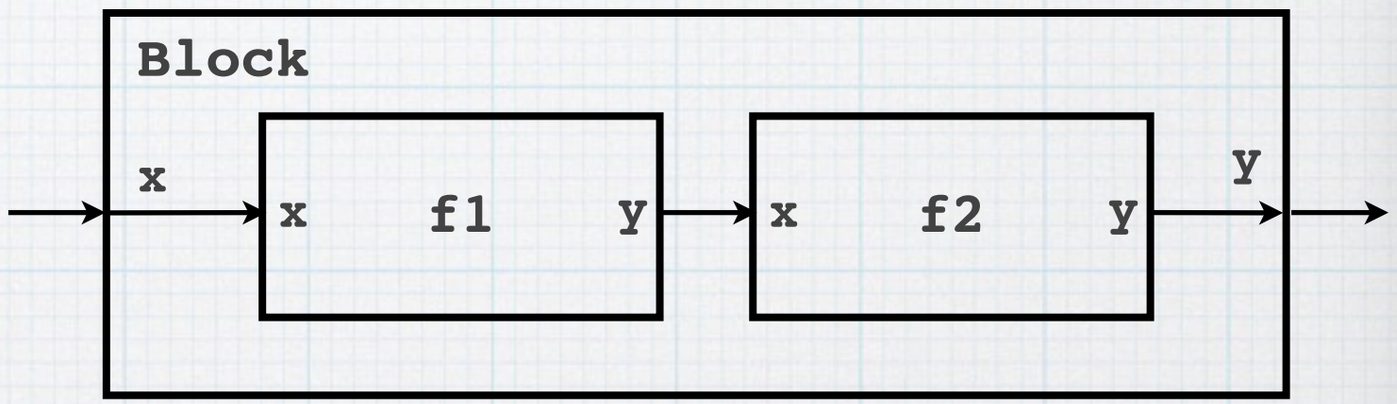
  - `FilterIO` aggregates other bundles.
  - "`flip`" recursively changes the "gender" of members.

Interfaces and Bulk Connections (2)

- Bundles help with making connections

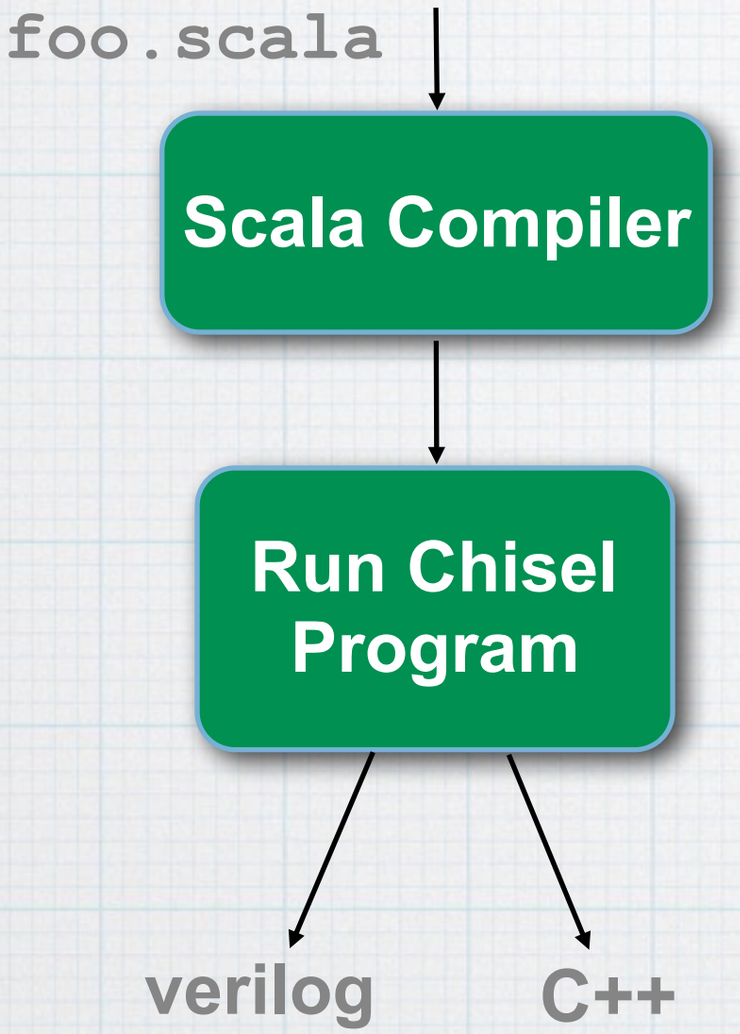
```
class Filter extends Module {
  val io = new FilterIO()
...
}
```


```
/ Bulk connections
class Block extends Module {
  val io = new FilterIO()

  val f1 = new Filter()
  val f2 = new Filter()

  f1.io.x <> io.x
  f1.io.y <> f2.io.x
  f2.io.y <> io.y
}
```



- "<>" bulk connects bundles of opposite gender, connecting leaf ports of the same name to each other.
- "<>" also promotes child component interfaces to parent component interfaces.

Running and Testing (1)



Scala Compiler generates an executable (Chisel program)

Execution of the Chisel program:

- generates an internal data structure (graph of "cells")
- resolves wire widths
- checks connectivity
- generates target output (currently verilog or C++)

Actually multiple different verilog targets are possible, pure simulation, Verilog for ASIC mapping, Verilog for FPGA mapping

More Information

- We will use Chisel 2.2.x not 3.0
- chisel.eecs.berkeley.edu/documentation.html

Chisel Cheat Sheet

Version 0.5 (beta): May 22, 2015

Notation In This Document:
For Functions and Constructors:
Arguments given as `kwd:type` (name and type(s))
Arguments in brackets (`[...]`) are optional.
For Operators:
`c`, `x`, `y` are Chisel `Data`; `n`, `m` are Scala `Int`
`w(x)`, `w(y)` are the widths of `x`, `y` (respectively)
`minVal(x)`, `maxVal(x)` are the minimum or maximum possible values of `x`

Basic Chisel Constructs

Chisel Wire Operators:

`val x = UInt()`	Allocate `a` as wire of type `UInt()`
`x := y`	Assign (connect) wire `y` to wire `x`
`x <> y`	Connect `x` and `y`, wire directionality is automatically inferred

When executes blocks conditionally by `Bool`, and is equivalent to Verilog `if`

```
when(condition1) {
  // run if condition1 true and skip rest
} .elsewhen(condition2) {
  // run if condition2 true and skip rest
} .unless(condition3) {
  // run if condition3 false and skip rest
} .otherwise {
  // run if none of the above ran
}
```

Switch executes blocks conditionally by data

```
switch(x) {
  is(value1) {
    // run if x === value1
  } is(value2) {
    // run if x === value2
  }
}
```

Enum generates value literals for enumerations

```
val s1::s2:: ... ::sn::Nil
    = Enum(nodeType:UInt, n:Int)
```

`s1`, `s2`, ..., `sn`	will be created as `nodeType` literals with distinct values
`nodeType`	type of `s1`, `s2`, ..., `sn`
`n`	element count

Math Helpers:

`log2Up(in:Int): Int`	log_2(`in`) rounded up
`log2Down(in:Int): Int`	log_2(`in`) rounded down
`isPow2(in:Int): Boolean`	`True` if `in` is a power of 2

Basic Data Types

Constructors:

```
Bool([x:Boolean])
Bits/UInt/SInt([x:Int/String], [width:Int])
```

`x`	*(optional)* create a literal from Scala type/ pased `String`, or declare unassigned if missing
`width`	*(optional)* bit width (inferred if missing)

Bits, UInt, SInt Casts: reinterpret cast except for:

`UInt` → `SInt`	Zero-extend to SInt

Bool Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	1
`x && y`	Logical AND	1
`x || y`	Logical OR	1

Bits Operators:

Chisel	Explanation	Width
`x(n)`	Extract bit, 0 is LSB	1
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y)`
`x >> y`	Dynamic right shift	`w(x) - minVal(y)`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate `x`, `n` times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If `c`, then `x`; else `y`	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	1
`x != y`	Inequality	1
`andR(x)`	AND-reduce	1
`orR(x)`	OR-reduce	1
`xorR(x)`	XOR-reduce	1

UInt, SInt Operators: (bitwdths given for `UInts`)

Chisel	Explanation	Width
`x + y`	Addition	`max(w(x), w(y))`
`x - y`	Subtraction	`max(w(x), w(y))`
`x * y`	Multiplication	`w(x) + w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y) - 1)`
`x > y`	Greater than	1
`x >= y`	Greater than or equal	1
`x < y`	Less than	1
`x <= y`	Less than or equal	1
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

State Elements

Registers retain state until updated

```
val my_reg = Reg([outType:Data], [next:Data],
                 [init:Data])
```

`outType`	*(optional)* register type (or inferred)
`next`	*(optional)* update value every clock
`init`	*(optional)* initialization value on reset

Updating: assign to latch new value on next clock:
`my_reg := next_val`
The last update (lexically, per clock) runs

Read-Write Memory provide addressable memories

```
val my_mem = Mem(out:Data, n:Int,
                 seqRead:Boolean)
```

`out`	memory element type
`n`	memory depth (elements)
`seqRead`	only update reads on clock edge

Using: access elements by indexing:
`val readVal = my_mem(addr:UInt/Int)`
for synchronous read: assign output to `Reg`
`mu_mem(addr:UInt/Int) := y`

Modules

Defining: subclass `Module` with elements, code:

```
class Accum(width:Int) extends Module {
  val io = new Bundle {
    val in = UInt(INPUT,  width)
    val out  = UInt(OUTPUT, width)
  }
  val sum = new Reg(UInt())
  sum := sum + io.in
  io.out := sum
}
```

Usage: access elements using dot notation:
(code inside a `Module` is always running)

```
val my_module = Module(new Accum(32))
my_module.io.in := some_data
val sum := my_module.io.out
```

Hardware Generation

Functions provide block abstractions for code
Defining: write Scala functions with Chisel code:

```
def Adder(op_a:UInt, op_b:UInt): UInt = {
  op_a + op_b
}
```

Usage: hardware is instantiated when called:

```
sum := Adder(UInt(1), some_data)
```

If/For can be used to control hardware generation and is equivalent to Verilog `generate if/for`

End of HDLs/Chisel Introduction

Advanced Chisel Later:

Memory Blocks
Polymorphism and Parameterization
Higher-order Functions