# CS250
# VLSI Systems Design

## Lecture 4: Chisel Introduction, Part 2

Fall 2011

Krste Asanović, John Wawrzynek
with
John Lazzaro and Brian Zimmer (TA)

Chisel was principally designed and created by Jonathan Bachrach



## Outline

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

# Update on Literal Constructors

- **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
Fix(5)          // signed decimal 4-bit literal of type Fix
Fix(-8)         // negative decimal 4-bit literal of type Fix
UFix(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
Fix(-5, 32)     // 32-bit decimal literal of type Fix, sign-extended
```

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



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

- Note: Vec is not a memory array. It's a collection of wires.
- 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) { Fix(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(dir = 'output)
  val data = Bits(width = 32, dir = 'input)
  val enq =  Bool(dir = 'input)
}
```

- The direction of an object can also be assigned at instantiation time:

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

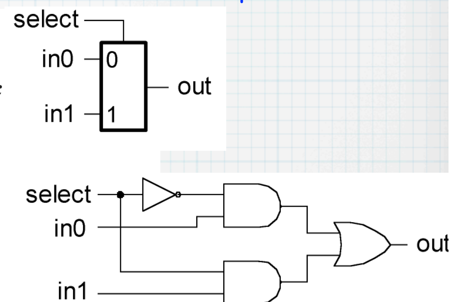
- The methods asInput and asOutput force all components of the data object to the requested direction.
- We'll see later, other methods exist for "flipping" direction, etc.



# Components

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

```
class Mux2 extends Component {
  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);
}
```

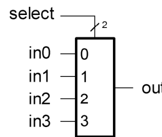


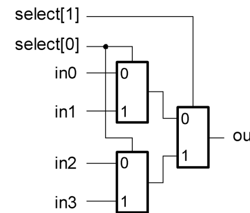

- The Component slot `io` is used to hold the interface definition, of type Bundle. `io` is assigned a Bundle that defines its ports.
- In this example,
  - `io` is assigned to an **anonymous** Bundle,
  - ":=" assignment operator, in Chisel wires the input of LHS to the output of circuit on the RHS

# Component Instantiation

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

```
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(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;
}
```





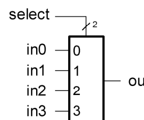

# Component Functional Abstraction

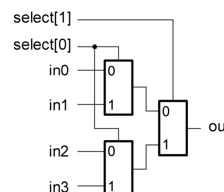
- Functional constructors for components simplify your code.

```
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

- **Construct 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: UFix, max: UFix) =
  Mux(n > max, UFix(0), n)

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

- Construct a circuit to output a pulse every **n** cycles:

```
// Produce pulse every n cycles.
def pulse(n: UFix) = counter(n - UFix(1)) === UFix(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: UFix) = 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() { UFix(width = 16) };
  when (c === 0) {
    r <== r + Ufix(1);
  }
```

- register r is updated on the next rising-clock-edge only iff c is zero.
- The argument to when is a predicate circuit expression that returns a Bool.
- The update block can only contain update statements using the update operator <==, simple expressions, and named wires defined with val.
- Cascaded conditional updates are prioritized from top to bottom in order of their Scala evaluation. For example,

```
when (c1)          { r <== Bits(1) }
when (c2)          { r <== Bits(2) }
when (Bool(true)) { r <== Bits(3) }
```

- Leads to:

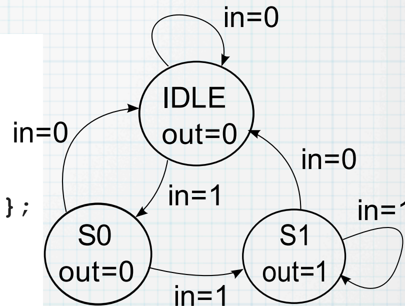
| c1 | c2 | r |
|---|---|---|
| 0 | 0 | 3 |
| 0 | 1 | 2 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

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

# Finite State Machine Specification (1)

- When blocks help in FSM specification:

```
class MyFSM extends Component {
  val io = new Bundle {
    val in  = Bool(dir = 'input);
    val out = Bool(dir = 'output);
  }
  val IDLE :: S0 :: S1 :: Nil = Enum(3){UFix()};
  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;
}
```



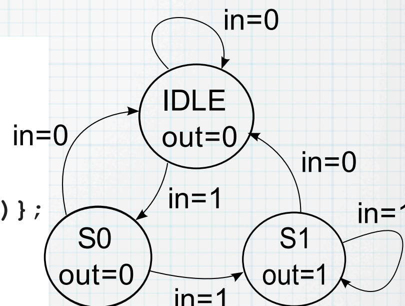

- Enum(3) generates three UFix lits, used here to represent states values.
- See tutorial for more complex FSM example.



# Finite State Machine Specification (2)

- Switch helps in FSM specification:

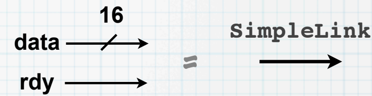
```
class MyFSM extends Component {
  val io = new Bundle {
    val in = Bool(dir = 'input);
    val out = Bool(dir = 'output);
  }
  val IDLE :: S0 :: S1 :: Nil = Enum(3) {UFix()};
  val state = Reg(resetVal = IDLE);
  switch (state) {
    is (IDLE) {
      when (io.in) { state <== S0 };
    }
    is (S0) {
      when (io.in) { state <== S1 };
      otherwise { state <== IDLE };
    }
    is (S1) {
    unless (io.in) { state <== IDLE };
    }
  }
  io.out := state === S1;
}
```



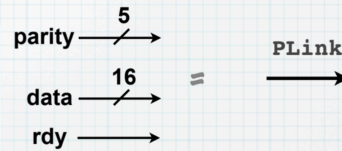
# Interfaces and Bulk Connections (1)

- **Bundles help with interface definitions**

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

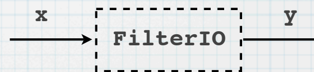



```
// Bundle Inheritance
class PLink extends SimpleLink {
  val parity = Bits(width=5, dir='output);
}
```




- `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.

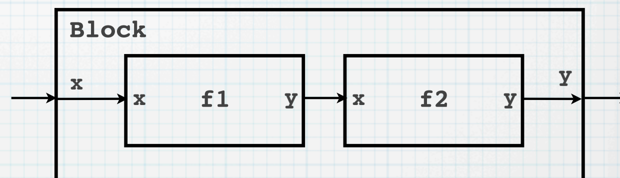


# Interfaces and Bulk Connections (2)

- **Bundles help with making connections**

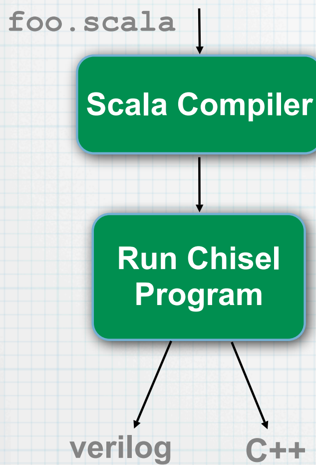
```
class Filter extends Component {
  val io = new FilterIO();
...
}
```

```
/ Bulk connections
class Block extends Component {
  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.
- "^^" 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



# Running and Testing (2)

- **Chisel translates into either Verilog or a C++ cycle-simulator, based on args.**
- Calling `chiselMain` results in building a circuit:

```
object tutorialMain {
  def main(args: Array[String]) = {
    chiselMain(args, () => new Mux2());
  }
}
```

- Scala looks for a top level object with "`main`"
- 2nd argument to ChiselMain is a anonymous function that creates the component under test.
- Chisel has a mechanism for testing circuits by providing test inputs and printing out results. Uses anonymous functions bound to named optional arguments.

```
object tutorialMain {
  def main(args: Array[String]) = {
    chiselMain(args, () => new Mux2(),
    scanner = (c:Mux2) => Scanner("%x %x %x", c.io.select,
                                       c.io.in0, c.io.in1),
    printer = (c:Mux2) => Printer("%= %= %= %=", c.io.select, c.io.in0,
                                          c.io.in1, c.io.out));
}
```

- first three hex numbers from each line are read in from standard input are bound to the select, in0, and in1 ports of the multiplexor circuit instance
- multiplexor circuit ports are printed out in hex format

# Running and Testing (3)

- Can test the multiplexor by creating a test file containing:

```
0 0 0 0
0 0 1 0
0 1 0 1
0 1 1 1
1 0 0 0
1 0 1 1
1 1 0 0
1 1 1 1
```

- Simple BSD shell command line tools can help.  Example:

```
$ cut -f 1,2,3 -d " " < test | mux > test.out
$ diff test test.out
```



# End of HDLs/Chisel Introduction part 2

Chisel Tutorial on website.

Advanced Chisel Later:

Memory Blocks
Polymorphism and Parameterization
Higher-order Functions