Chisel Tutorial

Jonathan Bachrach, Krste Asanović, John Wawrzynek
EECS Department, UC Berkeley
{jrb|krste|johnw}@eecs.berkeley.edu

November 19, 2011

1 Introduction

This document is a tutorial introduction to *Chisel* (Constructing Hardware In a Scala Embedded Language). Chisel is a hardware construction language embedded in the high-level programming language Scala. At some point we will provide a proper reference manual, in addition to more tutorial examples. In the meantime, this document along with a lot of trial and error should set you on your way to using Chisel. Chisel is really only a set of special class definitions, predefined objects, and usage conventions within Scala, so when you write a Chisel program you are actually writing a Scala program. However, for the tutorial we don't presume that you understand how to program in Scala. We will point out necessary Scala features through the Chisel examples we give, and significant hardware designs can be completed using only the material contained herein. But as you gain experience and want to make your code simpler or more reusable, you will find it important to leverage the underlying power of the Scala language. We recommend you consult one of the excellent Scala books to become more expert in Scala programming.

Chisel is still in its infancy and you are likely to encounter some implementation bugs, and perhaps even a few conceptual design problems. However, we are actively fixing and improving the language, and are open to bug reports and suggestions. Even in its early state, we hope Chisel will help designers be more productive in building designs that are easy to reuse and maintain.

Through the tutorial, we format commentary on our design choices as in this paragraph. You should be able to skip the commentary sections and still fully understand how to use Chisel, but we hope you'll find them interesting.

We were motivated to develop a new hardware language by years of struggle with existing hardware description languages in our research projects and hardware design courses. Verilog and VHDL were developed as hardware simulation *languages, and only later did they become a basis for hardware* synthesis*. Much of the semantics of these languages are not appropriate for hardware synthesis and, in fact, many constructs are simply not synthesizable. Other constructs are non-intuitive in how they map to hardware implementations, or their use can accidently lead to highly inefficient hardware structures. While it is possible to use a subset of these languages and yield acceptable results, they nonetheless present a cluttered and confusing specification model, particularly in an instructional setting.*

However, our strongest motivation for developing a new hardware language is our desire to change the way that electronic system design takes place. We believe that it is important to not only teach students how to design circuits, but also to teach them how to design circuit generators*—programs that automatically generate designs from a high-level set of design parameters and constraints. Through circuit generators, we hope to leverage the hard work of design experts and raise the level of design abstraction for everyone. To express flexible and scalable circuit construction, circuit generators must employ sophisticated programming techniques to make decisions concering how to best customize their output circuits according to high-level parameter values and constraints. While Verilog and VHDL include some primitive constructs for programmatic circuit generation, they lack the powerful facilities present in modern programming languages, such as object-oriented programming, type inference, support for functional programming, and reflection.*

Instead of building a new hardware design language from scratch, we chose to embed hardware construction primitives within an existing language. We picked Scala not only because it includes the programming features we feel are important for building circuit generators, but because it was specifically developed as a base for domain-specific languages.

2 Hardware expressible in Chisel

The initial version of Chisel only supports the expression of synchronous RTL (Register-Transfer Level) designs, with a single common clock. Synchronous RTL circuits can be expressed as a hierarchical composition of modules containing combinational logic and clocked state elements. Although Chisel assumes a single global clock, local clock gating logic is automatically generated for every state element in the design to save power.

Modern hardware designs often include multiple islands of logic, where each island uses a different clock and where islands must correctly communicate across clock island boundaries. Although clock-crossing synchronization circuits are notoriously difficult to design, there are known good solutions for most scenarios, which can be packaged as library elements for use by designers. As a result, most effort in new designs is spent in developing and verifying the functionality within each synchronous island rather than on passing values between islands.

In its current form, Chisel can be used to describe each of the synchronous islands individually. Existing tool frameworks can tie together these islands into a complete design. For example, a separate outer simulation framework can be used to model the assembly of islands running together. It should be noted that exhaustive dynamic verification of asynchronous communications is usually impossible and that more formal static approaches are usually necessary.

This version of Chisel also only supports binary logic, and does not support tri-state signals.

We focus on binary logic designs as they constitute the vast majority of designs in practice. We omit support for tri-state logic in the current Chisel language as this is in any case poorly supported by industry flows, and difficult to use reliably outside of controlled hard macros.

3 Datatypes in Chisel

Chisel datatypes are used to specify the type of values held in state elements or flowing on wires. While hardware designs ultimately operate on vectors of binary digits, other more abstract representations for values allow clearer specifications and help the tools generate more optimal circuits. In Chisel, a raw collection of bits is represented by the `Bits` type. Signed and unsigned integers are considered subsets of fixed-point numbers and are represented by types `Fix` and `UFix` respectively. Signed fixed-point numbers, including integers, are represented using two's-complement format. Boolean values are represented as type `Bool`. Note that these types are distinct from Scala's builtin types such as `Int`. Additionally, Chisel defines *Bundles* for making collections of values with named fields (similar to *structs* in other languages), and *Vecs* for indexable collections of values. Bundles and Vecs will be covered later.

Constant or literal values are expressed using Scala integers or strings passed to constructors for the types:

```
Bits(1)       // decimal 1-bit literal from Scala Int.
Bits("ha")    // hexadecimal 4-bit literal from string.
Bits("o12")   // octal 4-bit literal from string.
Bits("b1010") // binary 4-bit literal from string.

Fix(5)        // signed decimal 4-bit literal from Scala Int.
Fix(-8)       // negative decimal 4-bit literal from Scala Int.
UFix(5)       // unsigned decimal 3-bit literal from Scala Int.

Bool(true)    // Bool literals from Scala literals.
Bool(false)
```

Underscores can be used as separators in long string literals to aid readability, but are ignored when creating the value, e.g.:

```
Bits("h_dead_beef")   // 32-bit literal of type Bits
```

By default, the Chisel compiler will size each constant to the minimum number of bits required to hold the constant, including a sign bit for signed types. Bit widths can also be specified explicitly on literals, as shown below:

```
Bits("ha", 8)     // hexadecimal 8-bit literal of type Bits
Bits("o12", 6)    // octal 6-bit literal of type Bits
Bits("b1010", 12) // binary 12-bit literal of type Bits

Fix(5, 7)         // signed decimal 7-bit literal of type Fix
UFix(5, 8)        // unsigned decimal 8-bit literal of type UFix
```

For literals of type `Bits` and `UFix`, the value is zero-extended to the desired bit width. For literals of type `Fix`, the value is sign-extended to fill the desired bit width. If the given bit width is too small to hold the argument value, then a Chisel error is generated.

We are working on a more concise literal syntax for Chisel using symbolic prefix operators, but are stymied by the limitations of Scala operator overloading and have not yet settled on a syntax that is actually more readable than constructors taking strings.

We have also considered allowing Scala literals to be automatically converted to Chisel types, but this can cause type ambiguity and requires an additional import.

The Fix and UFix types will also later support an optional exponent field to allow Chisel to automatically produce optimized fixed-point arithmetic circuits.

4 Combinational Circuits

A circuit is represented as a graph of nodes in Chisel. Each node is a hardware operator that has zero or more inputs and that drives one output. A literal, introduced above, is a degenerate kind of node that has no inputs and drives a constant value on its output. One way to create and wire together nodes is using textual expressions. For example, we could express a simple combinational logic circuit using the following expression:

```
(a & b) | (~c & d)
```

The syntax should look familiar, with `&` and `|` representing bitwise-AND and -OR respectively, and `~` representing bitwise-NOT. The names `a` through `d` represent named wires of some (unspecified) width.

Any simple expression can be converted directly into a circuit tree, with named wires at the leaves and operators forming the internal nodes. The final circuit output of the expression is taken from the operator at the root of the tree, in this example, the bitwise-OR.

Simple expressions can build circuits in the shape of trees, but to construct circuits in the shape of arbitrary directed acyclic graphs (DAGs), we need to describe fan-out. In Chisel, we do this by naming a wire that holds a subexpression that we can then reference multiple times in subsequent expressions. We name a wire in Chisel by declaring a variable. For example, consider the select expression, which is used twice in the following multiplexer description:

```
val sel = a | b
val out = (sel & in1) | (~sel & in0)
```

The keyword `val` is part of Scala, and is used to name variables that have values that won't change. It is used here to name the Chisel wire, `sel`, holding the output of the first bitwise-OR operator so that the output can be used multiple times in the second expression.

5 Builtin Operators

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

Example	Explanation
Bitwise operators. Valid on Bits, Fix, UFix, Bool.	
`val invertedX = ~x`	Bitwise-NOT
`val hiBits = x & Bits("h_ffff_0000")`	Bitwise-AND
`val flagsOut = flagsIn | overflow`	Bitwise-OR
`val flagsOut = flagsIn ^ toggle`	Bitwise-XOR
Bitwise reductions. Valid on Bits, Fix, and UFix. Returns Bool.	
`val allSet = andR(x)`	AND-reduction
`val anySet = orR(x)`	OR-reduction
`val parity = xorR(x)`	XOR-reduction
Equality comparison. Valid on Bits, Fix, UFix, and Bool. Returns Bool.	
`val equ = x === y`	Equality
`val neq = x != y`	Inequality
Shifts. Valid on Bits, Fix, and UFix.	
`val twoToTheX = Fix(1) << x`	Logical left shift.
`val hiBits = x >> 16`	Right shift (logical on Bits & UFix, arithmetic on Fix).
Bitfield manipulation. Valid on Bits, Fix, UFix, and Bool.	
`val xLSB = x(0)`	Extract single bit, LSB has index 0.
`val xTopNibble = x(15,12)`	Extract bit field from end to start bit position.
`val usDebt = Fill(3, Bits("hA"))`	Replicate a bit string multiple times.
`val float = Cat(sign,exponent,mantissa)`	Concatenates bit fields, with first argument on left.
Logical operations. Valid on Bools.	
`val sleep = !busy`	Logical NOT.
`val hit = tagMatch && valid`	Logical AND.
`val stall = src1busy || src2busy`	Logical OR.
`val out = Mux(sel, inTrue, inFalse)`	Two-input mux where sel is a Bool.
Arithmetic operations. Valid on Nums: Fix and UFix.	
`val sum = a + b`	Addition.
`val diff = a - b`	Subtraction.
`val prod = a * b`	Multiplication.
`val div = a / b`	Division.
`val mod = a % b`	Modulus
Arithmetic comparisons. Valid on Nums: Fix and UFix. Returns Bool.	
`val gt = a > b`	Greater than.
`val gte = a >= b`	Greater than or equal.
`val lt = a < b`	Less than.
`val lte = a <= b`	Less than or equal.

5.1 Bitwidth Inference

Users are required to set bitwidths of ports and registers, but otherwise, bit widths on wires are automatically inferred unless set manually by the user. The bit-width inference engine starts from the graph's input ports and calculates node output bit widths from their respective input bit widths according to the following set of rules:

operation	bit width
`z = x + y`	`wz = max(wx, wy) + 1`
`z = x - y`	`wz = max(wx, wy) + 1`
`z = x & 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)`

where for instance wz is the bit width of wire z, and the `&` rule applies to all bitwise logical operations.

The bit-width inference process continues until no bit width changes. Except for right shifts by known constant amounts, the bit-width inference rules specify output bit widths that are never smaller than the input bit widths, and thus, output bit widths either grow or stay the same. Furthermore, the width of a register must be specified by the user either explicitly or from the bitwidth of the reset value. From these two requirements, we can show that the bit-width inference process will converge to a fixpoint.

Our choice of operator names was constrained by the Scala language. We have to use triple equals === for equality to allow the native Scala equals operator to remain usable.

We are also planning to add further operators that constrain bitwidth to the larger of the two inputs.

6 Functional Abstraction

We can define functions to factor out a repeated piece of logic that we later reuse multiple times in a design. For example, we can wrap up our earlier example of a simple combinational logic block as follows:

```
def clb(a: Bits, b: Bits, c: Bits, d: Bits): Bits = (a & b) | (~c & d)
```

where `clb` is the function which takes `a`, `b`, `c`, `d` as arguments and returns a wire to the output of a boolean circuit. The `def` keyword is part of Scala and introduces a function definition. The type of each argument is given after the semicolon, and the type the function returns is given after the semicolon that follows the argument list. The equals (=) sign separates the function argument list from the function definition.

We can then use the block in another circuit as follows:

```
val out = clb(a,b,c,d)
```

We will later describe many powerful ways to use functions to construct hardware using Scala's functional programming support.

7 Bundles and Vecs

`Bundle` and `Vec` are classes that allow the user to expand the set of Chisel datatypes with aggregates of other types.

Bundles group together several named fields of potentially different types into a coherent unit, much like a `struct` in C. Users define their own bundles by defining a class as a subclass of `Bundle`:

```
class MyFloat extends Bundle {
  val sign =  Bool();
  val exponent = Bits(width = 8);
  val significand =  Bits(width = 23);
}

val x = new MyFloat()
val xs = x.sign
```

A Scala convention is to capitalize the name of new classes and we suggest you follow that convention in Chisel too. The `width` named parameter to the `Bits` constructor specificies the number of bits in the type.

Vecs create an indexable vector of elements, and are constructed as follows:

```
val myVec = Vec(5) { Fix(width = 23) } // Vector of 5 23-bit signed integers.

val reg3 = myVec(3) // Connect to one element of vector.
```

(Note that we have to specify the type of the `Vec` elements inside the trailing curly brackets, as we have to pass the bitwidth parameter into the `Fix` constructor.)

The set of primitive classes (`Bits`,`Fix`,`UFix`,`Bool`) plus the aggregate classes (`Bundles` and `Vecs`) all inherit from a common superclass, `Data`. Every object that ultimately inherits from `Data` can be represented as a bit vector in a hardware design.

Bundles and Vecs can be arbitrarily nested to build complex data structures:

```
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.
}
```

Note that the builtin Chisel primitive and aggregate classes do not require the `new` when creating an instance, whereas new user datatypes will. A Scala `apply` constructor can be defined so that a user datatype also does not require `new`, as described in Section 14.

8 Ports

Ports are used as interfaces to hardware components. A port is simply any `Data` object that has directions assigned to its members.

Chisel provides port constructors to allow a direction to be added (input or output) to an object at construct time. Primitive port constructors take the number of bits as the first argument (except booleans which are always one bit) and direction as the second argument, where the direction is the symbol `'input` or `'output`.

An example port declaration is as follows:

```
class FIFOInput extends Bundle {
  val rdy =  Bool(dir = 'output)
  val data = Bits(width = 32, dir = 'input)
  val enq =  Bool(dir = 'input)
}
```

After defining `FIFOInput`, it becomes a new type that can be used as needed for component interfaces or for named collections of wires.

The direction of an object can also be assigned at instantation 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.

By folding directions into the object declarations, Chisel is able to provide powerful wiring constructs described later.

9 Components

In Chisel, *components* are very similar to *modules* in Verilog, defining a hierarchical structure in the generated circuit. The hierarchical component namespace is accessible in downstream tools to aid in debugging and physical layout. A user-defined component is defined as a *class* which:

- inherits from `Component`,
- contains an interface stored in a port field named `io`, and

- wires together subcircuits in its constructor.

As an example, consider defining your own two-input multiplexer as a component:

```
class Mux2 extends Component {
  val io = new Bundle{
    val sel = 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.sel & io.in1) | (~io.sel & io.in0);
}
```

The wiring interface to a component is a collection of ports in the form of a `Bundle`. The interface to the component is defined through a field named `io`. For `Mux2`, `io` is defined as a bundle with four fields, one for each multiplexer port.

The `:=` assignment operator, used here in the body of the definition, is a special operator in Chisel that wires the input of left-hand side to the output of the right-hand side.

9.1 Component Hierarchy

We can now construct circuit hierarchies, where we build larger components out of smaller sub-components. For example, we can build a 4-input multiplexer component in terms of the `Mux2` component by wiring together three 2-input multiplexers:

```
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 sel = Bits(width = 2, dir = 'input);
    val out = Bits(width = 1, dir = 'output);
  }
  val m0 = new Mux2();
  m0.io.sel := io.sel(0); m0.io.in0 := io.in0; m0.io.in1 := io.in1;

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

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

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

We again define the component interface as `io` and wire up the inputs and outputs. In this case, we create three `Mux2` children components, using the Scala `new` keyword to create a new object. We then wire them up to one another and to the ports of the `Mux4` interface.

10 Running and Testing Examples

Now that we have defined components, we will discuss how we actually run and test a circuit. Chisel translates into either `C++` or Verilog. In order to build a circuit we need to call `chisel_main`:

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

Testing is a crucial part of circuit design. Testing can be challenging and thus in Chisel we provide a mechanism for testing circuits by providing test inputs and printing out results:

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

where the first three hex numbers from each line are read in from standard input are bound to the `sel`, `in0`, and `in1` inputs of the multiplexer circuit, and the multiplexer inputs and `out` are printed out in hex format.

Currently, the only supported print format is `"%="`, which corresponds to the entire multiword argument in hex format (regardless of its size). The scanner has no format string argument restrictions, but is currently limited to reading only the low word (e.g., 64 bits on 64 bit i86 architecture) of each multiword. Currently, scanners and printers are only support with the C++ backend.

A user can test the multiplexer 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
```

and can be compared using a script as follows

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

11 State Elements

The simplest form of state element supported by Chisel is a positive-edge-triggered register, which can be instantiated functionally as:

```
Reg(in)
```

This circuit has an output that is a copy of the input signal `in` delayed by one clock cycle. Note that 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 the current version of Chisel, clock and reset are global signals that are implicity included where needed.

Using registers, we can quickly define a number of useful circuit constructs. For example, a rising-edge detector that takes a boolean signal in and outputs true when the current value is true and the previous value is false is given by:

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

Counters are an important sequential circuit. To construct an up-counter that counts up to a maximum value, `max`, then wraps around back to zero (i.e., modulo `max+1`), we write:

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

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

The `wraparound` function represents the combinational circuit that calculates the next value of the count. The counter register is created in the `counter` function with a reset value of `0` (with width large enough to hold `max`), to which the register will be initialized when the global reset for the circuit is asserted. The `:=` assignment to `x` in `counter` wires the output of the wraparound circuit to the input of the register `x`. Note that when `x` appears on the right-hand side of an assigment, its output is referenced, whereas when on the right-hand side, its input is referenced.

Counters can be used to build a number of useful sequential circuits. For example, we can build a pulse generator by outputting true when a counter reaches zero:

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

A square-wave generator can then be toggled by the pulse train, toggling between true and false on each pulse:

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

// Square wave where each half cycle has given period.
def squareWave(period: UFix) = toggle(pulse(period));
```

11.1 Forward Declarations

Purely combinational circuits cannot have cycles between nodes, and Chisel will report an error if such a cycle is detected. Because they do not have cycles, combinational circuits can always be constructed in a feed-forward manner, by adding new nodes whose inputs are derived from nodes that have already been defined. Sequential circuits naturally have feedback between nodes, and so it is sometimes necessary to reference an output wire before the producing node has been defined. Because Scala evaluates program statements sequentially, we have provided a `Wire` constructor to provide declaration of a wire node that can be used immediately, but whose input will be set later. For example, in a simple CPU, we need to define the `pcPlus4` and `branchTarget` wires so they can be referenced before defined:

```
val pcPlus4      = Wire(){ UFix() };
val branchTarget = Wire(){ UFix() };
val pcNext       = Mux(io.ctl.pcSel, branchTarget, pcPlus4);
val pcReg        = Reg(data = pcNext, resetVal = UFix(0, 32));
pcPlus4         := pcReg + UFix(4);
...
branchTarget    := addOut;
```

The type given in braces following the `Wire` constructor allows the Scala compiler to know the types of `pcPlus4` and `branchTarget` before they are defined. The wiring operator `:=` is then used to wire up the connection after `pcReg` and `addOut` are defined.

11.2 Conditional Updates

In our previous examples using registers, we simply wired their inputs to combinational logic blocks. When describing the operation of state elements, it is often useful to instead specify when updates to the registers will occur and to specify these updates spread across several separate statements. Chisel provides conditional update rules in the form of the `when` construct to support this style of sequential logic description. For example,

```
val r = Reg() { UFix(16) };
when (c === UFix(0) ) {
  r <== r + UFix(1);
}
```

where register `r` is updated at the end of the current clock cycle only if `c` is zero. The argument to `when` is a predicate circuit expression that returns a `Bool`. The update block following `when` 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. In particular, the first conditional update whose condition evaluates to true is chosen to execute its updates. For example,

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

leads to `r` being updated according to the following truth table:

```
c1 c2  r
0   0  r  // r unchanged
0   1  2
1   0  1
1   1  1  // c1 takes precedence over c2.
```

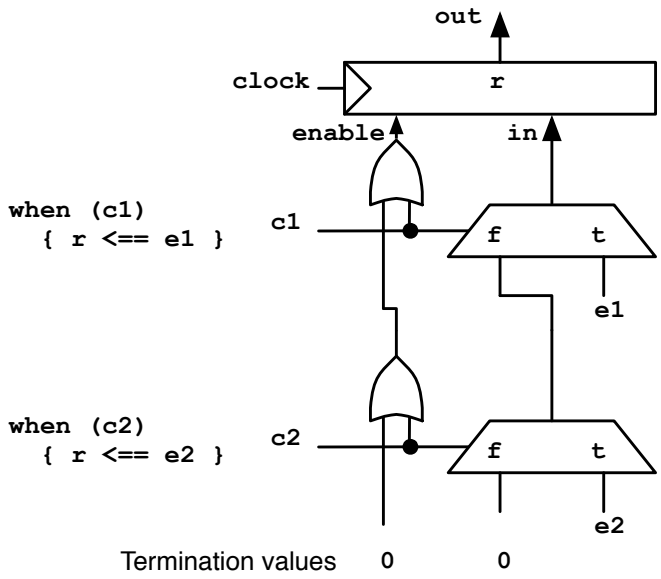

Figure 1: Equivalent hardware constructed for conditional updates. Each `when` statement adds another level of data mux and ORs the predicate into the enable chain. The compiler effectively adds the termination values to the end of the chain automatically.

Figure 1 shows how each conditional update can be viewed as adding a mux to the input of a register, where the mux select is controlled by the `when` predicate such that the update expression is muxed into the register when the predicate is true, else the input comes from following code. In addition, the predicate is OR-ed into a firing signal that drives the load enable of the register. Subsequent `when` constructs chain on to the input of the earlier muxes and on to the OR chain. The compiler adds termination values to the end of the chain so that if no conditional updates fire in a clock cycle, the load enable of the register will be deasserted and the register value will not change.

Chisel provides some syntactic sugar for other common forms of conditional update. The `unless` construct is the same as `when` but negates its condition. In other words,

```
unless (c) { body }
```

is the same as

```
when (!c) { body }
```

The `otherwise` construct is the same as `when` with a true condition. In other words,

```
otherwise { body }
```

is the same as

```
when (Bool(true)) { body }
```

The update block can target multiple registers, and there can be different overlapping subsets of registers present in different update blocks. Each register is only affected by conditions in which it appears. For example:

```
when (c1)   { r <== Fix(1); s <== Fix(1) }
when (c2)   { r <== Fix(2) }
otherwise   { r <== Fix(3); s <== Fix(3) }
```

leads to `r` and `s` being updated according to the following truth table:

```
c1 c2  r  s
0   0  3  3
0   1  2  3 // r updated in c2 block, s updated in otherwise block.
1   0  1  1
1   1  1  1
```

We are considering adding a different form of conditional update, where only a single update block will take effect. These atomic updates are similar to Bluespec guarded atomic actions.

Conditional update constructs can be nested and any given block is executed under the conjunction of all outer nesting conditions. For example,

```
when (a) { when (b) { body } }
```

is the same as:

```
when (a && b) { body }
```

We introduce the `switch` statement for conditional updates involving a series of comparisons against a common key. For example,

```
switch(idx) {
 is(v1)    { u1 };
 is(v2)    { u2 };
 otherwise { ud };
}
```

is equivalent to:

```
when (idx === v1) { u1 };
when (idx === v2) { u2 };
otherwise         { ud };
```

Chisel also allows a `Wire`, i.e., the output of some combinational logic, to be the target of conditional update statements to allow complex combinational logic expressions to be built incrementally. Chisel does not allow a combinational output to be incompletely specified and will report an error if an unconditional update is not encountered for a combinational output.

In Verilog, if a procedural specification of a combinational logic block is incomplete, a latch will silently be inferred causing many frustrating bugs.

It could be possible to add more analysis to the Chisel compiler, to determine if a set of predicates covers all possibilities. But for now, we require a single predicate that is always true in the chain of conditional updates to a `Wire`*.*

11.3 Finite State Machines

A common type of sequential circuit used in digital design is a Finite State Machine (FSM). An example of a simple FSM is a parity generator:

```
class Parity extends Component {
  val io = new Bundle {
    val in  = Bool(dir = 'input);
    val out = Bool(dir = 'output); }
  val s_even :: s_odd :: Nil = Enum(2){ UFix() };
  val state  = Reg(resetVal = s_even);
  when (s.in) {
    when (state === s_even) { state <== s_odd  };
    when (state === s_odd)  { state <== s_even };
  }
  io.out := (state === s_odd);
}
```

where `Enum(2){ UFix() }` generates two `UFix` literals and where the states are updated when `in` is true. It is worth noting that all of the mechanisms for FSMs are built upon registers, wires, and conditional updates.

Below is a more complicated FSM example which is a circuit for accepting money for a vending machine:

```
class VendingMachine extends Component {
  val io = new Bundle {
    val nickel = Bool(dir = 'input);
    val dime   = Bool(dir = 'input);
    val rdy    = Bool(dir = 'output); }
  val s_idle :: s_5 :: s_10 :: s_15 :: s_ok :: Nil = Enum(5){ UFix() };
  val state = Reg(resetVal = s_idle);
  when (state === s_idle) {
    when (io.nickel) { state <== s_5; }
    when (io.dime)   { state <== s_10; }
  }
  when (state === s_5) {
    when (io.nickel) { state <== s_10; }
    when (io.dime)   { state <== s_15; }
  }
  when (state === s_10) {
    when (io.nickel) { state <== s_15; }
    when (io.dime)   { state <== s_ok; }
  }
  when (state === s_15) {
    when (io.nickel) { state <== s_ok; }
    when (io.dime)   { state <== s_ok; }
  }
  when (state === s_ok) {
    state <== s_idle;
  }
  io.rdy := (state === s_ok);
}
```

Here is the vending machine FSM defined with `switch` statement:

```
class VendingMachine extends Component {
  val io = new Bundle {
    val nickel = Bool(dir = 'input);
    val dime   = Bool(dir = 'input);
    val rdy    = Bool(dir = 'output); }
  val s_idle :: s_5 :: s_10 :: s_15 :: s_ok :: Nil = Enum(5){ UFIx() };
  val state = Reg(resetVal = s_idle);
  switch (state) {
      is (s_idle) {
      when (io.nickel) { state <== s_5; }
      when (io.dime)   { state <== s_10; }
    } is (s_5) {
      when (io.nickel) { state <== s_10; }
      when (io.dime)   { state <== s_15; }
    } is (s_10) {
      when (io.nickel) { state <== s_15; }
      when (io.dime)   { state <== s_ok; }
    } is (s_15) {
      when (io.nickel) { state <== s_ok; }
```

```
      when (io.dime)   { state <== s_ok; }
    } is (s_ok) {
      state <== s_idle;
    }
  }
  io.rdy := (state === s_ok);
}
```

12 Memories

Chisel provides facilities for creating both read only and read/write memories.

12.1 ROM

Users can define read only memories with:

```
Rom(inits: Seq[Data])
```

where `inits` is a sequence of initial `Data` literals that initialize the ROM and define the element type. For example, users can create a small ROM initialized to `1, 2, 4, 8` and loop through all values using a counter as an address generator as follows:

```
val m = Rom(Array(UFix(1), UFix(2), UFix(4), UFix(8)));
val r = m.read(counter(UFix(3)));
```

We can create an `n` value sine lookup table using a ROM initialized as follows:

```
def sinTable (amp: Double, n: Int) = {
  val ts = Range(0, n, 1).map(i => (i*2*Pi)/(n.toDouble-1)  Pi);
  Rom(ts.map(t => Fix(round(amp * sin(t)))
}
def sinWave (amp: Double, n: Int) =
  sinTable(amp, n).read(counter(UFix(n))
```

where `amp` is used to scale the fixpoint values stored in the ROM.

12.2 RAM

Simple memories allow users to define simple single write multiple read memories. The basic creation of the memory object looks as follows:

```
Mem(depth: Int, wrEnable: Bool, wrAddr: UFix, wrData: Data)
```

where `n` is the number is the depth, `wrEnable` says whether writes occur, `wrAddr` provide the address to write to, and `wrData` give the data to write. This memory object can then be read from using the read method with a read address. For example, an audio recorder could be defined as follows:

```
def audioRecorder(n: Int) = {
  val addr = counter(UFix(n));
  Ram(n, button(), addr, mic()).read(Mux(button(), 0, addr))
}
```

where a counter is used as an address generator into a memory. The device records while the button is held down and plays back when it is released.

We can use simple memory to create register files. For example we can make a one write port, two read port register file with 32 registers as follows:

```
val regs = Mem(32, wr_en, wr_addr, wr_data);
val idat = regs.read(iaddr);
val mdat = regs.read(maddr);
```

where a new read port is created for each call to read.

An extended version of the `Mem` constructor, `Mem4`, is similar, with additional parameters to mimic common static memory (SRAM) behaviors. The following is an example of a memory with one read and write port:

```
Mem4.setDefaultReadLatency(1);                 // Default: 0
val regfile = Mem4(64, io.data_in1);           // A memory with no ports, with data width derived from another si
regfile.setHexInitFile("hex_init_values.txt"); // Data formatted for Verilog readmemh
regfile.write(io.addr_in, io.data_in1, io.wen, w_mask = io.bit_mask);
val read_data = regfile.read(io.addr_in, io.data_in1, oe = !io.wen, cs = Bool(true));
```

This example can be written more compactly as:

```
val regfile = Mem4(64, io.wen, io.addr_in, io.data_in1, w_mask = io.bit_mask,
 readLatency = 1, hexInitFile = "hex_init_values.txt");
val read_data = regfile(io.addr_in, io.data_in1, oe = !io.wen, cs = Bool(true));
```

By default, this memory will be compiled to Verilog RTL. To produce a reference to a verilog instance of a memory module, add `regfile.setTarget('inst)`. When Chisel compiles to Verilog, a second file will be generated, eg: `design.conf`, which can be used by the synthesis design flow to construct the requested memory objects.

In addition to read and write memory ports, the `Mem4` memory object supports combined read/write ports. The following example defines a memory with two read/write ports:

```
val rw_mem = Mem4(64, io.data_in1);
rw_mem.setReadLatency(1);
val rw_data_out = rw_mem.rw(io.addr_in, io.data_in1, io.wen);
io.data_out2 := rw_mem.rw(io.addr_in,
                          io.data_in2,
                          we = Bool(true),      // Active high write enable
                          w_mask = ~Bits(0,32), // Optional write mask
                          cs = Bool(true),      // Active high chip select
                          oe = Bool(true));     // Active high output enable
```

13 Interfaces and Bulk Connections

For more sophisticated components it is often useful to define and instantiate interface classes while defining component IO. First and foremost, interface classes promote reuse allowing users to capture once and for all common interfaces in a useful form. Secondly, interfaces allow users to dramatically reduce wiring by supporting *bulk connections* between producer and consumer components. Finally, users can make changes in large interfaces in one place reducing the number of updates required when adding or removing pieces of the interface.

13.1 Port Classes, Subclasses, and Nesting

As we saw earlier, users can define their own interfaces by defining a class that subclasses `Bundle`. For example, a user could define a simple link for handshaking data as follows:

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

We can then extend `SimpleLink` by adding parity bits using bundle inheritance:

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

In general, users can organize their interfaces into hierarchies using inheritance.

From there we can define a filter interface by nesting two `PLink`s into a new `FilterIO` bundle:

```
class FilterIO extends Bundle {
  val x = new PLink().flip;
  val y = new PLink();
}
```

where `flip` recursively changes the "gender" of a bundle, changing input to output and output to input.

We can now define a filter by defining a filter class extending component:

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

where the `io` field contains `FilterIO`.

13.2 Bundle Vectors

Beyond single elements, vectors of elements form richer hierarchical interfaces. For example, in order to create a crossbar with a vector of inputs, producing a vector of outputs, and selected by a UFix input, we utilize the `Vec` constructor:

```
class CrossbarIo(n: Int) extends Bundle {
  val in  = Vec(n){ new PLink().flip() };
  val sel = UFix(width = (log(n)/log(2)).toInt, dir = 'input);
  val out = Vec(n){ new PLink() };
}
```

where `Vec` takes a size as the first argument and a block returning a port as the second argument.

13.3 Bulk Connections

We can now compose two filters into a filter block as follows:

```
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;
}
```

where <> bulk connects interfaces of opposite gender, and ^^ promotes child component interfaces to parent component interfaces. Bulk connections connect leaf ports of the same name to each other. After all connections are made and the circuit is being elaborated, Chisel warns users if ports have other than exactly one connection to them.

13.4 Port Views

Consider a simple CPU consisting of control path and data path subcomponents and host and memory interfaces shown in Figure 2. In this CPU we can see that the control path and data path each connect only to a part of the instruction and data memory interfaces. Chisel allows users to do this with interface views without breaking an interface into unnatural subinterfaces. A user first defines the natural interface to say a ROM as follows:

```
class RomIo(view: List[String] = null) extends Bundle(view) {
  val isVal = Bool(dir = 'input);
  val raddr = UFix(width = 32, dir = 'input);
  val rdata = Bits(width = 32, dir = 'output);
}
```

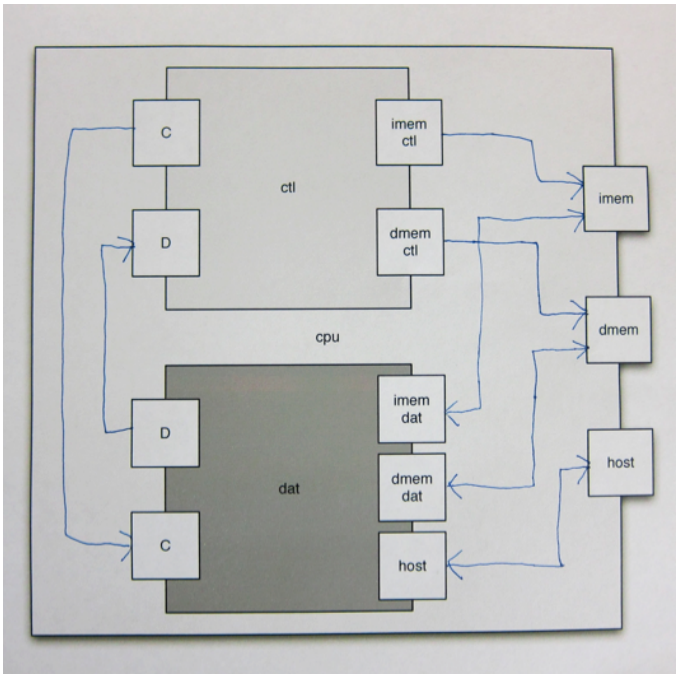

Figure 2: Simple CPU involving control and data path subcomponents and host and memory interfaces.

where `view` is provided as a constructor parameter that is relayed to the bundle superclass allowing views of `RomIo` to be constructed. From there, multiple ROM view constructors on the ROM interface can be constructed:

```
object RomIo {
  def RomCtlIo() = new RomIo(List("isVal"));
  def RomDatIo() = new RomIo(List("raddr", "rdata"));
}
```

for the control and data subsets of the interface. Views are specified by listing out the names of the interface fields that are to be included. Now the control path can build an interface in terms of memory views of their control lines:

```
class CpathIo extends Bundle() {
  val imem = RomCtlIo().flip();
  val dmem = RamCtlIo().flip();
  ...
}
```

We can now wire up the CPU using bulk connects of the bundle views as we would with other bundles:

```
class Cpu extends Component {
  val io = new CpuIo();
  val c  = new CtlPath();
  val d  = new DatPath();
  c.io.ctl  <> d.io.ctl;
  c.io.dat  <> d.io.dat;
  c.io.imem ^^ io.imem;
  d.io.imem ^^ io.imem;
  c.io.dmem ^^ io.dmem;
  d.io.dmem ^^ io.dmem;
  d.io.host ^^ io.host;
}
```

14 Functional Creation of Components

It is also useful to be able to make a functional interface for component construction. For instance, we could build a constructor that takes multiplexer inputs as parameters and returns the multiplexer output:

```
object Mux2 {
  def apply (sel: Bits, in0: Bits, in1: Bits) = {
    val m = new Mux2();
    m.io.in0 := in0;
    m.io.in1 := in1;
    m.io.sel := sel;
    m.io.out
  }
}
```

where `object Mux2` creates a Scala singleton object on the `Mux2` component class, and `apply` defines a method for creation of a `Mux2` instance. With this `Mux2` creation function, the specification of `Mux4` now is significantly simpler.

```
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 sel = Bits(width = 2, dir = 'input);
    val out = Bits(width = 1, dir = 'output);
  };
  io.out := Mux2(io.sel(1),
                 Mux2(io.sel(0), io.in0, io.in1),
                 Mux2(io.sel(0), io.in2, io.in3));
}
```

Selecting inputs is so useful that Chisel builds it in and calls it `Mux`. However, unlike `Mux2` defined above, the builtin version allows any datatype on `in0` and `in1` as long as they have a common super class. In Section 15 we will see how to define this ourselves.

Chisel provides `MuxCase` which is an n-way `Mux`

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

where each condition / value is represented as a tuple in a Scala array and where `MuxCase` can be translated into the following `Mux` expression:

```
Mux(c1, a, Mux(c2, b, Mux(..., default)));
```

Chisel also provides `MuxLookup` which is an n-way indexed multiplexer:

```
MuxLookup(idx, default, Array(UFix(0) -> a, UFix(1) -> b, ...))
```

which can be rewritten in terms of:`MuxCase` as follows:

```
MuxCase(default, Array((idx === UFix(0)) -> a, (idx === UFix(1)) -> b, ...))
```

Note that the cases (eg. c1, c2) must be in parentheses.

15 Polymorphism and Parameterization

Scala is a strongly typed language and uses parameterized types to specify generic functions and classes. In this section, we show how Chisel users can define their own reusable functions and classes using parameterized classes.

This section is advanced and can be skipped at first reading.

15.1 Parameterized Functions

Earlier we defined `Mux2` on `Bool`, but now we show how we can define a generic multiplexer function. We define this function as taking a boolean condition and con and alt arguments (corresponding to then and else expressions) of type `T`:

```
def Mux[T <: Bits](c: Bool, con: T, alt: T): T { ... }
```

where `T` is required to be a subclass of `Bits`. Scala ensures that in each usage of `Mux`, it can find a common superclass of the actual con and alt argument types, otherwise it causes a Scala compilation type error. For example,

```
Mux(c, UFix(10), UFix(11))
```

yields a `UFix` wire because the `con` and `alt` arguments are each of type `UFix`.

We now present a more advanced example of parameterized functions for defining an inner product FIR digital filter generically over Chisel `Num`'s. The inner product FIR filter can be mathematically defined as:

$$y[t] = \sum_j w_j * x_j[t-j] \tag{1}$$

where x is the input and w is a vector of weights. In Chisel this can be defined as:

```
def innerProductFIR[T <: Num] (w: Array[Int], x: T) =
  foldR(Range(0, w.length).map(i => Num(w(i)) * delay(x, i)), _ + _)

def delay[T <: Bits](x: T, n: Int): T =
  if (n == 0) x else Reg(delay(x, n - 1))

def foldR[T <: Bits] (x: Seq[T], f: (T, T) => T): T =
  if (x.length == 1) x(0) else f(x(0), foldR(x.slice(1, x.length), f))
```

where `delay` creates a `n` cycle delayed copy of its input and `foldR` (for fold right) constructs a reduction circuit given a binary combiner function `f`. In this case, `foldR` creates a summation circuit. Finally, the `innerProductFIR` function is constrained to work on inputs of type `Num` where Chisel multiplication and addition are defined.

15.2 Parameterized Classes

Like parameterized functions, we can also parameterize classes to make them more reusable. For instance, we can generalize the Filter class to use any kind of link. We do so by parameterizing the `FilterIO` class and defining the constructor to take a zero argument type constructor function as follow:

```
class FilterIO[T <: Data]()(type: => T) extends Bundle {
  val x = type.asInput.flip;
  val y = type.asOutput;
}
```

We can now define `Filter` by defining a component class that also takes a link type constructor argument and passes it through to the `FilterIO` interface constructor:

```
class Filter[T <: Data]()(type: => T) extends Component {
  val io = (new FilterIO()){ type };
  ...
}
```

We can now define a `PLink` based `Filter` as follows:

```
val f = (new Filter()){ new PLink() };
```

where the curly braces `{ }` denote a zero argument function (aka thunk) that in this case creates the link type.

A generic FIFO could be defined as shown in Figure **??** and used as follows:

```
class DataBundle() extends Bundle {
  val A = UFix(width = 32);
  val B = UFix(width = 32);
}

object FifoDemo {
  def apply () = (new Fifo(32)){ new DataBundle() };
}


class FifoIO[T <: Data]()(gen: => T) extends Bundle() {
  val enq_val = Bool(dir = 'input);
  val enq_rdy = Bool(dir = 'output);
  val deq_val = Bool(dir = 'output);
  val deq_rdy = Bool(dir = 'input);
  val enq_dat = gen.asInput;
  val deq_dat = gen.asOutput;
}

class Fifo[T <: Data] (n: Int)(gen: => T) extends Component {
  val io           = new FifoIO()( gen );
  val enq_ptr      = Reg(resetVal = UFix(0, sizeof(n)));
  val deq_ptr      = Reg(resetVal = UFix(0, sizeof(n)));
  val is_full      = Reg(resetVal = Bool(false));
  val do_enq       = io.enq_rdy && io.enq_val;
  val do_deq       = io.deq_rdy && io.deq_val;
  val is_empty     = !is_full && (enq_ptr === deq_ptr);
  val deq_ptr_inc  = deq_ptr + UFix(1);
  val enq_ptr_inc  = enq_ptr + UFix(1);
  val is_full_next =
    Mux(do_enq && ~do_deq && ( enq_ptr_inc === deq_ptr ), Bool(true),
    Mux(do_deq && is_full,                               Bool(false),
        is_full));
  enq_ptr <== Mux(do_enq, enq_ptr_inc, enq_ptr);
  deq_ptr <== Mux(do_deq, deq_ptr_inc, deq_ptr);
  is_full <== is_full_next;
  val ram = Mem(n, do_enq, enq_ptr, io.enq_dat);
  io.enq_rdy := !is_full;
  io.deq_val := !is_empty;
  ram.read(deq_ptr) ^^ io.deq_dat;
}
```

Figure 3: Fifo example.

16 Acknowlegements

Many people have helped out in the design of Chisel, and we thank them for their patience, bravery, and belief in a better way. Many Berkeley EECS students in the Isis group gave weekly feedback as the design evolved including but not limited to Yunsup Lee, Andrew Waterman, Scott Beamer, Chris Celio, etc. Yunsup Lee gave us feedback in response to the first RISC-V implementation, called TrainWreck, translated from Verilog to Chisel. Andrew Waterman and Yunsup Lee helped us get our Verilog backend up and running and Chisel TrainWreck running on an FPGA. Brian Richards was the first actual Chisel user, first translating (with Huy Vo) John Hauser's FPU Verilog code to Chisel, and later implementing generic memory blocks. Brian gave many invaluable comments on the design and brought a vast experience in hardware design and design tools. Chris Batten shared his fast multiword C++ template library that inspired our fast emulation library. Huy Vo became our undergraduate research assistant and was the first to actually assist in the Chisel implementation. We appreciate all the EECS students who participated in the Chisel bootcamp and proposed and worked on hardware design projects all of which pushed the Chisel envelope. We appreciate the work that James Martin and Alex Williams did in writing and translating network and memory controllers and non-blocking caches. Finally, Chisel's functional programming and

bit-width inference ideas were inspired by earlier work on a hardware description language called Gel [**?**] designed in collaboration with Dany Qumsiyeh and Mark Tobenkin.

References

[1] Bachrach, J., Qumsiyeh, D., Tobenkin, M. *Hardware Scripting in Gel*. in Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th.