Outline

- Background and History of Hardware Description
- Brief Introduction to Chisel Part 1
- Next Monday, Chisel Part 2
Design Entry

- Schematic entry/editing used to be the standard method in industry and universities.
- Used in commonly until ~2002

Schematics are intuitive. They match our use of gate-level or block diagrams.

Somewhat physical. They imply a physical implementation.

Require a special tool (editor).

Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow on large designs.

Hardware Description Languages (HDLs) are the new standard
- except for PC board design, where schematics are still used.

Hardware Description Languages

- Originally invented for simulation.
- Now "logic synthesis" tools exist to automatically convert from HDL source to circuits.
- High-level constructs greatly improves designer productivity.
- However, this may lead you to falsely believe that hardware design can be reduced to writing programs!*

Basic Idea: language constructs describe circuits with two basic forms:

- **Structural descriptions**: connections of components. Nearly one-to-one correspondence to with schematic diagram.

- **Behavioral descriptions**: use high-level constructs (similar to conventional programming) to describe the circuit function.

“Structural” example:
Decoder(output x0,x1,x2,x3; inputs a,b)
wire abar, bbar;
inv(bbar, b);
inv(abar, a);
and(x0, abar, bbar);
and(x1, abar, b);
and(x2, a, bbar);
and(x3, a, b);
}

“Behavioral” example:
Decoder(output x0,x1,x2,x3; inputs a,b)
{
case [a b]
  00: [x0 x1 x2 x3] = 0x1;
  01: [x0 x1 x2 x3] = 0x2;
  10: [x0 x1 x2 x3] = 0x4;
  11: [x0 x1 x2 x3] = 0x8;
endcase;
}

Warning: this is a fake HDL!

*Describing hardware with a language is similar, however, to writing a parallel program.
Standard Design Methodology

Hierarchically defines structure and/or function of circuit.

HDL Specification

Simulation

Synthesis

Verification: Does the design behave as required with regards to function (and timing, and power consumption)?

Maps specification to resources of implementation platform (FPGA or ASIC).

Note: This is not the entire story. Other tools are useful for analyzing HDL specifications. More on this later.

HDL History

- 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 1364 (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: Synfora PICO, AutoESL)
Verilog “Issues”

- 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 leads to confusion (particularly for beginners), and bugs.

- 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.

```verilog
module gray2bin1 (bin, gray);
parameter SIZE = 8;
output [SIZE-1:0] bin;
input [SIZE-1:0] gray;
genvar i;
genenerate
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 (later support for UTL 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.
- First tutorial document available online. Formal reference later.
## Chisel Acknowledgements

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jonathan Bachrach</td>
<td>Principal developer</td>
</tr>
<tr>
<td>Huy Vo</td>
<td>Undergrad research assistant, Chisel Developer</td>
</tr>
<tr>
<td>Brian Richards</td>
<td>First actual user (translated FPU Verilog code to Chisel)</td>
</tr>
<tr>
<td>Yunsup Lee, Andrew Waterman</td>
<td>Early users for processor design mapped to FPGAs</td>
</tr>
<tr>
<td>Scott Beamer</td>
<td>Early user continues to write chisel routers and give feedback</td>
</tr>
<tr>
<td>Chris Celio</td>
<td>Pushed the design, given feedback, and written the most Chisel code in writing RiscV code</td>
</tr>
<tr>
<td>Chris Batten</td>
<td>Fast C++ template library that inspired Chisel fast simulator</td>
</tr>
<tr>
<td>James Martin and Alex Williams</td>
<td>Work on writing and translating network and memory controllers and non-blocking caches</td>
</tr>
<tr>
<td>Anonymous</td>
<td>Participants in first Chisel bootcamp</td>
</tr>
</tbody>
</table>

---

## Simple Combinational Logic Example

```plaintext
// 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 are "bitwise".

- 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:

```scala
// 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` will be generated at one place in the circuit and then "fanned-out" to other places where `out` appears.

```scala
// 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:

```scala
// 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".

```scala
// Constructing multiple copies
val z = (x & XOR(x,y)) | (XOR(x,y) & y)
```

- Functions wrapping up simple logic are light-weight. This results hierarchy in your code, but no hierarchy in the Chisel output.

- Later we'll see how to write polymorphic functions.

- We'll see later that Chisel Components 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:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>Raw collection of bits</td>
</tr>
<tr>
<td>Fix</td>
<td>Signed fixed-point number</td>
</tr>
<tr>
<td>UFix</td>
<td>Unsigned fixed-point number</td>
</tr>
<tr>
<td>Bool</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

- Signed and unsigned integers are special cases of Fix and UFix, respectively.
- 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:

  ```scala
  // 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:

  ```scala
  // 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):

```scala
class FIFOInput extends Bundle {
  val rdy = Bool('output) // Indicates if FIFO has space
  val data = Bits(32, 'input) // 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 "components" ...

Literals

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

```scala
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("-5") // 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:

```scala
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.
### Builtin Operators (1)

**Bitwise operators. Valid on Bits, Fix, UFix, Bool.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val invertedX = ~x</code></td>
<td>Bitwise-NOT</td>
</tr>
<tr>
<td><code>val hiBits = x &amp; Bits(&quot;h_ffff_0000&quot;)</code></td>
<td>Bitwise-AND</td>
</tr>
<tr>
<td>`val flagsOut = flagsIn</td>
<td>overflow`</td>
</tr>
<tr>
<td><code>val flagsOut = flagsIn ^ toggle</code></td>
<td>Bitwise-XOR</td>
</tr>
</tbody>
</table>

**Bitwise reductions. Valid on Bits, Fix, and UFix. Returns Bool.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val allSet = and(x)</code></td>
<td>AND-reduction</td>
</tr>
<tr>
<td><code>val anySet = or(x)</code></td>
<td>OR-reduction</td>
</tr>
<tr>
<td><code>val parity = xor(x)</code></td>
<td>XOR-reduction</td>
</tr>
</tbody>
</table>

**Equality comparison. Valid on Bits, Fix, UFix, and Bool. Returns Bool.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val equ = x === y</code></td>
<td>Equality</td>
</tr>
<tr>
<td><code>val neq = x !== y</code></td>
<td>Inequality</td>
</tr>
</tbody>
</table>

**Shifts. Valid on Bits, Fix, and UFix.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val twoToTheX = Fix('1') &lt;&lt; x</code></td>
<td>Logical left shift.</td>
</tr>
<tr>
<td><code>val hiBits = x &gt;&gt; 16</code></td>
<td>Logical right shift.</td>
</tr>
</tbody>
</table>

### Builtin Operators (2)

**Bitfield manipulation. Valid on Bits, Fix, UFix, and Bool.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val xLSB = x(0)</code></td>
<td>Extract single bit, LSB has index 0.</td>
</tr>
<tr>
<td><code>val xTopNibble = x(15,12)</code></td>
<td>Extract bit field from start to end bit position. Repeat.</td>
</tr>
<tr>
<td><code>val usDebt = Fill(3, Bits(&quot;hA'&quot;))</code></td>
<td>Concatenates bit fields, with first argument on left.</td>
</tr>
<tr>
<td><code>val float = Cat(exponent,mantissa)</code></td>
<td></td>
</tr>
</tbody>
</table>

**Logical operations. Valid on Booleans.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val sleep = !busy</code></td>
<td>Logical NOT.</td>
</tr>
<tr>
<td><code>val hit = tagMatch &amp;&amp; valid</code></td>
<td>Logical AND.</td>
</tr>
<tr>
<td>`val stall = src1busy</td>
<td></td>
</tr>
<tr>
<td><code>val out = Mux(sel, inTrue, inFalse)</code></td>
<td>Two-input mux where sel is a Bool.</td>
</tr>
</tbody>
</table>

**Arithmetic operations. Valid on Nums: Fix and UFix.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val sum = a + b</code></td>
<td>Addition.</td>
</tr>
<tr>
<td><code>val diff = a - b</code></td>
<td>Subtraction AND.</td>
</tr>
<tr>
<td><code>val prod = a * b</code></td>
<td>Multiplication.</td>
</tr>
<tr>
<td><code>val div = a / b</code></td>
<td>Division.</td>
</tr>
<tr>
<td><code>val mod = a % b</code></td>
<td>Modulus.</td>
</tr>
</tbody>
</table>

**Arithmetic comparisons. Valid on Nums: Fix and UFix. Returns Bool.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val gt = a &gt; b</code></td>
<td>Greater than.</td>
</tr>
<tr>
<td><code>val gte = a &gt;= b</code></td>
<td>Greater than or equal.</td>
</tr>
<tr>
<td><code>val lt = a &lt; b</code></td>
<td>Less than.</td>
</tr>
<tr>
<td><code>val lte = a &lt;= b</code></td>
<td>Less than or equal.</td>
</tr>
</tbody>
</table>
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 components) and registers must be specified, but otherwise widths are inferred with the application of the following rules:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Width Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z = x + y$</td>
<td>$w_z = \text{max}(w_x, w_y) + 1$</td>
</tr>
<tr>
<td>$z = x - y$</td>
<td>$w_z = \text{max}(w_x, w_y) + 1$</td>
</tr>
<tr>
<td>$z = x &lt;\text{bitwise-op}&gt; y$</td>
<td>$w_z = \text{max}(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = \text{Mux}(c, x, y)$</td>
<td>$w_z = \text{max}(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = w * y$</td>
<td>$w_z = w_x + w_y$</td>
</tr>
<tr>
<td>$z = x &lt;&lt; n$</td>
<td>$w_z = w_x + \text{maxNum}(n)$</td>
</tr>
<tr>
<td>$z = x &gt;&gt; n$</td>
<td>$w_z = w_x - \text{minNum}(n)$</td>
</tr>
<tr>
<td>$z = \text{Cat}(x, y)$</td>
<td>$w_z = w_x + w_y$</td>
</tr>
<tr>
<td>$z = \text{Fill}(n, x)$</td>
<td>$w_z = w_x * \text{maxNum}(n)$</td>
</tr>
</tbody>
</table>

- Bit-width inference is still under discussion. We might be adding more operators that preserve input widths (truncate to max of the input widths).

---

End of HDLs/Chisel Introduction part 1

Chisel Tutorial on website later today.

More Introduction in Class Monday:
- Components and Circuit Hierarchy
- More on Muxes
- Registers
- Conditional Update Rules
- FSMs
- Memory Blocks
- More on Bundles, Arrays and Bulk Connections