Chisel: Constructing Hardware in a Scala Embedded Language

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee,
Andrew Waterman, Rimas Avižienis, John Wawrzynek, Krste Asanović
EECS Department, UC Berkeley *
{jrb|huytbvo|richards|yunsup|waterman|rimas|johnw|krste}@eecs.berkeley.edu

ABSTRACT

In this paper we introduce *Chisel*, a new hardware construction language that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. By embedding Chisel in the Scala programming language, we raise the level of hardware design abstraction by providing concepts including object orientation, functional programming, parameterized types, and type inference. Chisel can generate a high-speed C++-based cycle-accurate software simulator, or low-level Verilog designed to map to either FPGAs or to a standard ASIC flow for synthesis. This paper presents Chisel, its embedding in Scala, hardware examples, and results for C++ simulation, Verilog emulation and ASIC synthesis.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: [Design Aids – automatic synthesis, hardware description languages]

General Terms

Design, Languages, Performance

Keywords

CAD

1. INTRODUCTION

The dominant traditional hardware-description languages (HDLs), Verilog and VHDL, were originally developed as hardware *simulation* languages, and were only later adopted as a basis for hardware *synthesis*. Because the semantics of these languages are based around simulation, synthesizable designs must be inferred from a subset of the language, complicating tool development and designer education. These languages also lack the powerful abstraction facilities that are common in modern software languages, which leads to low designer productivity by making it difficult to reuse components. Constructing efficient hardware designs requires extensive design-space exploration of alternative system microarchitectures [9] but these traditional HDLs have limited module generation facilities and are ill-suited to producing and composing the highly parameterized module generators required to support thorough design-space exploration. Recent extensions such as SystemVerilog improve the type system and parameterized generate facilities but still lack many powerful programming language features.

To work around these limitations, one common approach is to use another language as a macro processing language for an underlying HDL. For example, Genesis2 uses Perl to provide more flexible parameterization and elaboration of hardware blocks written in SystemVerilog [9]. The language called Verischemelog [6] provides a Scheme syntax for specifying modules in a similar format to Verilog. JHDL [1] equates Java classes with modules. HML [7] uses standard ML functions to wire together a circuit. These approaches allow familiar and powerful languages to be macro languages for hardware netlists, but effectively require leaf components of the design to be described in the underlying HDL. This combined approach is cumbersome, combining the poor abstraction facilities of the underlying HDL with a completely different high-level programming model that does not understand hardware types and semantics.

An alternative approach is to begin from a domain-specific application programming language from which a hardware block is generated. Esterel [2] uses event-based statements to program hardware for reactive systems. DIL [4] is an intermediate language targeted at stream processing and hardware virtualization. Bluespec [3] supports a general concurrent computation model, based on guarded atomic actions. While these can provide great designer productivity when the task in hand matches the pattern encoded in the application programming model, they are a poor match for tasks outside their domain. For example, the design of a programmable microprocessor is not well described in a stream programming model, and guarded atomic actions are not a natural way to express a high-level DSP algorithm. Furthermore, in general it is difficult to derive an efficient microarchitecture from a higher-level computation model, especially if the goal is a programmable engine to run many applications, where the human designer would prefer to write a

*Research supported by DoE Award DE-SC0003624, and by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

generator to explore this design space in detail.

In this paper, we introduce Chisel (Constructing Hardware In a Scala Embedded Language), a new hardware design language we have developed based on the Scala programming language [8]. Chisel is intended to be a simple platform that provides modern programming language features for accurately specifying low-level hardware blocks, but which can be readily extended to capture many useful high-level hardware design patterns. By using a flexible platform, each module in a project can employ whichever design pattern best fits that design, and designers can freely combine multiple modules regardless of their programming model. Chisel can generate fast cycle-accurate C++ simulators for a design, or generate low-level Verilog suitable for either FPGA emulation or ASIC synthesis with standard tools. We present several design examples and results from emulation and synthesis experiments.

2. CHISEL OVERVIEW

Instead of building a new hardware design language from scratch, we chose to embed hardware construction primitives within the Scala programming language. We chose Scala for a number of reasons: Scala 1) is a very powerful language with features we feel are important for building circuit generators, 2) is specifically developed as a base for domain-specific languages, 3) compiles to the JVM, 4) has a large set of development tools and IDEs, and 5) has a fairly large and growing user community. Chisel comprises a set of Scala libraries that define new hardware datatypes and a set of routines to convert a hardware data structure into either a fast C++ simulator or low-level Verilog for emulation or synthesis. This section describes the features of the base Chisel system, whereas the next two sections describe how Chisel supports abstraction and powerful generators.

2.1 Chisel Datatypes

The basic Chisel datatypes are used to specify the type of values held in state elements or flowing on wires. 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. Boolean values are represented as type `Bool`. Note that these types are distinct from Scala's builtin types such as `Int`. Constant or literal values are expressed using Scala integers or strings passed to constructors for the types.

Chisel provides a *Bundle* class, which the user extends to make collections of values with named fields (similar to *structs* in other languages):

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

The keyword `val` is part of Scala, and is used to name variables that have values that won't change. The `width` named parameter to the `UFix` constructor specifies the number of bits in the type. Chisel also provides *Vecs* for indexable collections of values:

```
// Vector of five 23-bit signed integers.
val myVec = Vec(5) { Fix(width = 23) }
val reg3 = myVec(3) // Connect to one element of vector.
```

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

2.2 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 is a degenerate 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:

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

where `&` and `|` represent bitwise-AND and -OR respectively, and `~` represents 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 must describe fan-out. In Chisel, we can name a wire holding a subexpression by declaring a variable, then referencing it multiple times in subsequent expressions:

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

The named Chisel wire `sel` holds the output of the first bitwise-OR operator so that the output can be used multiple times in the second expression.

Bit widths 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, always preserving exact results unless an explicit truncation is requested.

2.3 Functions

We can define functions to factor out a repeated piece of logic that we later reuse multiple times in a design:

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

The `def` keyword is part of Scala and introduces a function definition, with each argument followed by a colon then its type, and the function return type given after the colon following the argument list. The equals (`=`) sign indicates the start of the function definition.

2.4 Ports and Components

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

```
class FIFOInput extends Bundle {
  val ready = Bool(OUTPUT)
  val valid = Bool(INPUT)
  val data  = Bits(32, INPUT)
}
```

`FIFOInput` becomes a new type that can be used in component interfaces or for named collections of wires.

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
}
```

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

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: (1) inherits from `Component`, (2) contains an interface stored in a port field named `io`, and (3) wires together subcircuits in its constructor. As an example, consider defining a two-input multiplexer as a component:

```
class Mux2 extends Component {
  val io = new Bundle {
    val sel = Bits(1, INPUT)
    val in0 = Bits(1, INPUT)
    val in1 = Bits(1, INPUT)
    val out = Bits(1, 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`, held in a field named `io`. 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.

Port classes represent the interface to a component, and users can organize interfaces into hierarchies using standard Scala facilities. For example, a user could define a simple link for handshaking data as follows:

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

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

```
class PLink extends SimpleLink {
  val parity = Bits(5, OUTPUT)
}
```

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`.

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(ceilLog2(n), 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.

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 between components. 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.

2.5 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 implicitly 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)
```

2.6 Conditional Updates

In the previous examples with 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(width = 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`.

In a sequence of conditional updates, the last conditional update whose condition is true takes priority. 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	2	c2 takes precedence over c1

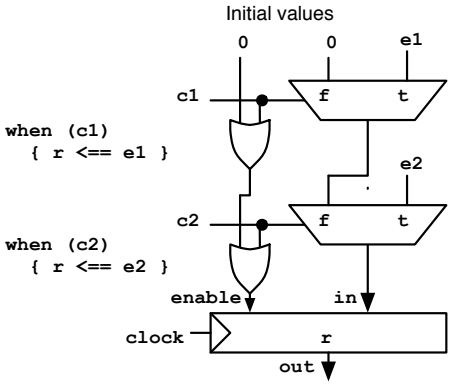

Figure 1: Equivalent hardware constructed for conditional updates.

Figure 1 shows how each conditional update can be viewed as inserting a mux before the input of a register to select either the update expression or the previous input according to the `when` predicate. In addition, the predicate is OR-ed into a firing signal that drives the load enable of the register. The compiler places initialization values at the beginning 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.

3. ABSTRACTION

In this section we discuss abstraction within Chisel. Abstraction is an important aspect of Chisel as it 1) allows users to conveniently create reusable objects and functions, 2) allows users to define their own data types, and 3) allows users to better capture particular design patterns by writing their own domain-specific languages on top of Chisel.

3.1 Polymorphism and Parameterized Types

Scala is a strongly typed language and uses parameterized types to specify generic functions and classes. Chisel users can define their own reusable functions and classes using parameterized classes. For instance we can write a parameterized function for defining an inner-product FIR digital filter generically over Chisel `Num`'s. The inner product FIR filter can be mathematically defined as:

$$y[t] = \sum_j w_j * x_j[t-j] \quad (1)$$

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 an `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` for which Chisel multiplication and addition are defined.

Like parameterized functions, we can also parameterize classes to make them more reusable. For instance, we can generalize the Filter class, defined in section 2.4, 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 follows:

```
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 }
  ...
}
```

3.2 Abstract Data Types

Through support for abstract data types, Chisel permits much simpler code than would otherwise be possible. For example, consider constructing a block, such as the FFT, requiring arithmetic on complex numbers. In Chisel, complex numbers can be defined as follows:

```
class Complex(val real: Fix, val imag: Fix) extends Bundle {
  def +(b: Complex): Complex =
    new Complex(real + b.real, imag + b.imag)
  ...
}
```

where we overload infix operators to provide an intuitive algebraic interface. Complex numbers can now be used in both the interface and in arithmetic:

```
class Example extends Component {
  val io = new Bundle {
    val a = new Complex(Fix(2, INPUT), Fix(2, INPUT))
    val b = new Complex(Fix(2, INPUT), Fix(2, INPUT))
    val out = new Complex(Fix(2, OUTPUT), Fix(2, OUTPUT))
  }
  val c = io.a + io.b
  io.out.r := c.r
  io.out.i := c.i
}
```

4. GENERATORS

A key motivation for embedding Chisel in Scala is to support highly parameterized circuit generators, a weakness of traditional HDLs.

4.1 Cache Generator

One example of a highly parameterized subsystem is a memory cache generator. In Chisel, the basic configuration options can first be defined:

```
object CacheParams {
  val DIR_MAPPED  = 0
  val SET_ASSOC   = 1
  val WRITE_THRU  = 0
  val WRITE_BACK  = 1
}
```

The main body of the cache generator component can then be declared with desired generator parameters and optional default values. The `io` bundle then references two IO interface bundles, one specifying a connection to a CPU and the other defining the memory interface. Computed parameters are then defined, followed by the main body of the generator:

```
class Cache(cache_type: Int    = DIR_MAPPED,
            associativity: Int = 1,
            line_size: Int     = 128,
            cache_depth: Int   = 16,
            write_policy: Int  = WRITE_THRU
           ) extends Component {
  val io = new Bundle() {
    val cpu = new IoCacheToCPU()
    val mem = new IoCacheToMem().flip()
  }
  val addr_idx_width = ( log(cache_depth) / log(2) ).toInt
  val addr_off_width = ( log(line_size/32) / log(2) ).toInt
  val addr_tag_width = 32 - addr_idx_width - addr_off_width - 2
  val log2_assoc     = ( log(associativity) / log(2) ).toInt
   ...
  if (cache_type == DIR_MAPPED)
    ...
}
```

The resulting `Cache` generator can then be used in a larger system:

```
  ...
  val data_cache = new Cache(cache_type = SET_ASSOC, line_size = 64)
  connection_to_cpu <> data_cache.io.cpu
  connection_to_mem <> data_cache.io.mem
  ...
```

4.2 Sorting Network

In addition to offering flexible parameterization, Chisel supports recursive creation of hardware subsystems. In the example below a simple sorting network is specified using a two-input `SortBlock` defined with handshaking ports. First, a simple queue IO interface data type is defined by extending the `Bundle` class. This data type will be used to define connections between the sorting primitives:

```
  class IoSortBlockOut extends Bundle() {
    val output     = Bits(sort_data_size, OUTPUT)
    val output_rdy = Bool(OUTPUT)
    val has_output = Bool(OUTPUT)
    val pop        = Bool(INPUT)
  }
```

The `SortBlock` primitive is then defined to output the minimum of the two inputs, subject to handshaking:

```
  class SortBlock extends Component() {
    override val io = new Bundle() {
      val in1 = new IoSortBlockOut()
      val in2 = new IoSortBlockOut()
      val out = new IoSortBlockOut()
    }
    ...
  }
```

Using this sorting primitive, it is then possible to define a recursive architecture to find the minimum of a vector of numbers. `SortVector` below recursively finds the minimum of the first and second halves of the input vector, and returns the minimum of the two results. This example also demonstrates the power of using `Bundle` to combine inputs and outputs along with arrays of `Bundle` using `Vec`.

```
  class SortVector(in_width: Int) extends Component() {
    val io = new Bundle() {
      val in_vec = Vec(in_width) { new IoSortBlockOut().flip }
      val out    = new IoSortBlockOut()
    }
    val min1 = new SortPair()
    min1.io.out <> io.out
    val midpoint = in_width / 2
    if (in_width < 4) {
      // Connect first input directly to min1
      min1.io.in1 <> io.in_vec(0)
    } else {
      val min_first_half = new SortVector(midpoint)
      for (i <- 0 until midpoint)
        min_first_half.io.in_vec(i) <> io.in_vec(i)
      min1.io.in1 <> min_first_half.io.out
    }
    if (in_width < 3) {
      min1.io.in2 <> io.in_vec(1)
    } else {
      val min_second_half = new SortVector(in_width - midpoint)
      for (i <- midpoint until in_width)
        min_second_half.io.in_vec(i - midpoint) <> io.in_vec(i)
      min1.io.in2 <> min_second_half.io.out
    }
  }
```

Note that Verilog is not able to describe this type of recursion, and a designer would need to use a different language, such as `Python`, to generate Verilog from a recursive routine.

4.3 Memory

Memories are given special treatment in Chisel since hardware implementations of memory have many variations, e.g., FPGA memories are instantiated quite differently from ASIC memories. Chisel defines a memory abstraction that can map to either simple Verilog behavioral descriptions, or to instances of memory modules that are available from external memory generators provided by foundry or IP vendors. In the simplest form, Chisel allows memory to be defined with a single write port and multiple read ports as follows:

```
Mem(depth: Int,
    target: Symbol = 'default, readLatency: Int = 0)
```

where `depth` is the number of memory locations, `target` is the type of memory used, `readLatency` is the latency of read ports to be defined on the memory. A memory object can then be read from using the `read(rdAddress)` method. For example, an audio recorder could be defined as follows:

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

where a counter is used as an address generator into a memory. The device records while `button` is `true`, or plays back when `false`.

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)
regs.write(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.

Additional parameters are available to mimic common memory behaviors, to aid with the process of mapping to real-world hardware. The following is an example of a memory that is first defined with no memory ports, after which read, write, or read/write ports are added:

```
val regfile =
  Mem(64, readLatency = 1,
      hexInitFile = "hex_init_values.txt");
regfile.write(addr_in, data_in1, wen, w_mask = bit_mask);
val read_data = regfile.read(addr_in);
```

By default, this memory will be compiled to Verilog RTL. To produce a reference to a Verilog instance of a memory module, one adds `target = 'inst` to the constructor call. When Chisel compiles to Verilog, a second file will be generated, e.g., `design.conf`, which can be used by the synthesis design flow to construct the requested memory objects.

5. FAST C++ SIMULATOR

Fast simulation is crucial to reduce hardware development time. Custom logic simulation engines can provide fast cycle-accurate simulation, but are generally too expensive to be used by individual designers. FPGA emulation approaches are valuable but the FPGA tool flow can take hours to map a design iteration. Conventional software Verilog RTL simulators are popular, as they can be run by individual designers on workstations or shared server farms, but are slow.

For Chisel, we have developed a fast C++ simulator for RTL debugging. The Chisel compiler produces a C++ class for each Chisel design, with a C++ interface including clock-low and clock-high methods. We rely on two techniques to accelerate execution. First, the simulator code generation strategy is based on a templated C++ multi-word bit-vector runtime library that executes all the basic Chisel operators. The C++ templates specialize operations for bit vectors using a two-level template scheme that is first parameterized on bits and then on words. In particular, all overhead is removed for the case where the RTL bit vector fits into the host machine's native word size. Second, we remove as much branching as possible in the code so that we can best utilize the ILP available in a modern processor and minimize the number of stalls.

6. RESULTS

In this section, we present preliminary results on using Chisel for various hardware designs. To measure designer productivity, we took a simple 3-stage 32-bit RISC processor that was originally hand-written in Verilog, and converted it to equivalent Chisel code. The original Verilog code was 3020 lines of code whereas the resulting Chisel code was only 1046 lines, yielding a nearly 3× reduction.

To compare quality of results, we used a set of floating-point primitive components we have designed in Chisel, including multiplication, addition, and several data conversion operators. A 64-bit Fused-Multiply-Add (FMA) unit has been mapped to both Verilog and C++ emulation code, and both results have been simulated in testbenches using SoftFloat and TestFloat [5] to verify IEEE-754-2008 compliance. The generated Verilog was mapped to a commercial 65 nm process and compared to the same design described using hand-coded Verilog, and as expected there was no significant difference in results:

Source	Clock Period	Total Area	Logic Area
Chisel	7ns	62197 um^2	60801 um^2
Verilog	7ns	62881 um^2	61485 um^2
Chisel	2.5ns, Retimed	66472 um^2	61279 um^2
Verilog	2.5ns, Retimed	67034 um^2	62227 um^2

To compare the speed of simulation using the Chisel C++ simulator, we used a more sophisticated 64-bit five-stage in-order RISC pipeline with a floating-point unit, MMU, and caches. We compared the speed of Chisel C++ simulation and Synopsys VCS Verilog simulation when booting a research OS on this processor (88, 291, 350 cycles total) with results as follows:

Simulator	Time (s)	Speedup
VCS RTL simulator	5390	1.00
Chisel C++ RTL simulator	694	7.77

The Chisel-generated C++ simulator is approximately 8× faster than VCS.

Finally, we have developed a complete 64-bit vector processor including FPUs, MMUs, and 32 K 4-way set-associative instruction and data caches. The Chisel code was used to generate an LVS and DRC-clean GDSII layout in an IBM 45 nm SOI 10-metal layer process using memory-compiler-generated 6T and 8T SRAM blocks. Total area was 1.76 mm^2, with a critical path of 1 ns.

7. CONCLUSION

Chisel makes the power of a modern software programming language available for hardware design, supporting high-level abstractions and parameterized generators without mandating a particular computational model, while also providing high-quality Verilog RTL output and a fast C++ simulator.

The Chisel system and hardware libraries are being made available as an open-source project available at:

`http://chisel.eecs.berkeley.edu`

to encourage wide adoption.

8. ACKNOWLEDGEMENTS

We'd like to thank Christopher Batten for sharing his fast multiword C++ template library that inspired our fast emulation library. We'd also like to thank all the Berkeley EECS graduate students who participated in the Chisel bootcamp and have given feedback on Chisel after using it in various classes and research projects.

9. REFERENCES

[1] Bellows, P., and Hutchings, B. JHDL – an HDL for reconfigurable systems. *IEEE Symposium on FPGAs for Custom Computing Machines* (1998).

[2] Berry, G., and Gonthier, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 10*, 2 (1992).

[3] Bluespec Inc. *Bluespec(tm) SystemVerilog Reference Guide: Description of the Bluespec SystemVerilog Language and Libraries*. Waltham, MA, 2004.

[4] Goldstein, S., and Budiu, M. Fast compilation for pipelined reconfigurable fabrics. *ACM/FPGA Symposium on Field Programmable Gate Arrays* (1999).

[5] Hauser, J. The softfloat and testfloat packages. *http://www.jhauser.us/arithmetic/index.html*.

[6] Jenning, J., and Beuscher, E. Verischemelog: Verilog embedded in scheme. *Proceedings of DSL'99: The 2nd conference on Domain Specific Languages* (Oct 1999).

[7] Li, Y., and Leeser, M. HML – a novel hardware description language and its translation to VHDL. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8*, 1 (Oct 2000).

[8] Odersky, M. e. a. Scala programming language. *http://www.scala-lang.org/*.

[9] Shacham, O., Azizi, O., Wachs, M., Qadeer, W., Asgar, Z., Kelley, K., Stevenson, J., Solomatnikov, A., Firoozshahian, A., Lee, B., Richardson, S., and M., H. Rethinking digital design: Why design must change. *IEEE Micro* (Nov/Dec 2010).

10. SUPPLEMENTAL

In this section we give more detailed examples, results, and discussion of Chisel.

10.1 Builtin Operators

Chisel defines a set of hardware operators for the builtin types which can be found in Table 1.

10.2 Layers of Languages

Scala was designed to support the creation of embedded domain-specific languages. In fact, it is easy to create a series of languages, one layered on top of another, resulting in improved clarity and efficiency in specification. As a small example, we can easily build a `switch` statement involving a series of comparisons against a common key, based on the Chisel conditional updates introduced earlier.

As a small example, we can easily build a `switch` statement involving a series of comparisons against a common key, based on the Chisel conditional updates introduced earlier.

switch construct	translates into

`switch(idx) {`
`  is(v1) { u1 }`
`  is(v2) { u2 }`
`}`
	`when (idx === v2) { u2 }`
	`when (idx === v1) { u1 }`

The `switch` construct supports simple specification of FSMs:

```
val s_even :: s_odd :: Nil = Enum(2){ UFix() }
val state = Reg(resetVal = s_even)
switch (s.in) {
  is (s_even) { state <== s_odd  }
  is (s_odd)  { state <== s_even }
}
```

We are exploring embedding new domain-specific languages in Chisel to provide high-level behavioral synthesis.

10.3 Scala Embedding Discussion

Embedding Chisel in Scala gave a number of advantages but also presented a number of challenges.

In Scala, we are able to cleanly integrate Chisel components, bundles and interfaces with Scala classes. Using introspection, we can find all relevant fields and their names in Scala objects. Scala also provides a number of facilities for writing domain-specific languages including operator overloading.

Unfortunately, there are other areas where it is still challenging to customize the language seamlessly. The first one is providing a succinct literal format. Unfortunately, unlike Common Lisp, in Scala it is impossible to define new tokens. The second one is that, at least in standard Scala, it is impossible to overload existing syntax, such as `if` statements. In general, there is no way to extend the Scala syntax in arbitrary ways. Higher-order functions and lightweight thunks help, but the result is that the Chisel syntax is slightly more awkward than we'd ideally like.

Yet another challenge is providing informative error messages. When errors occur, it is possible to provide stack backtraces to report to users on what line number an error occurred. Unfortunately, it is challenging to filter the stack trace to give the user the exact line the error occurred.

Although Scala has a large number of data types, we are not able to completely layer our hardware data types on to these Scala ones. We instead built a parallel type hierarchy. Scala has a very powerful parameterized type system that allows us to create generic functions and classes that can be precisely type checked. Unfortunately, the type system is not able to infer bit widths automatically, so we have to add a separate bit-width inference pass, as described below. The advantage is that our Chisel design is more portable to other host languages.

10.4 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.

10.5 BlackBox's

Users can create wrappers for existing opaque IP components using `BlackBox`es which are Components with only IO and no body. For example, a Verilog-based memory controller module can be linked in by defining it as a subclass of `BlackBox`:

```
class MemoryController extends BlackBox {
  val io = new MemoryIo();
}
```

and then by instantiating it and connecting to it as done with any other Chisel component. The emitted Verilog will then contain code to create and wire in the module.

10.6 Vending Machine FSM Example

Here is an example of a vending machine FSM defined with a `switch` statement:

```
class VendingMachine extends Component {
  val io = new Bundle {
    val nickel = Bool(INPUT)
    val dime   = Bool(INPUT)
    val rdy    = Bool(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 }
```

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 >> UFix(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.

Table 1: Chisel operators on builtin data types.

```
      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)
}
```

10.7 Simulation Performance

In Section 6 we compared simulation speed for a 64-bit five-stage RISC processor design using a Chisel-generated C++ simulator and Synopsys VCS Verilog simulation. Table 2 is a more complete breakdown of the results in terms of compile time, run time, and total time. We also include results for a Chisel-generated FPGA emulation, which provides the fastest per-cycle emulation performance but with a large compile time.

Because of compilation time, the fastest backend for simulation performance depends on the number of target cycles to be simulated. While the Chisel C++ emulator runs approximately 10× faster than VCS, as shown in Figure 2, this advantage is only realized when simulating millions of cycles or more. FPGA emulation is only fastest for simulations exceeding billions of target cycles. We are planning to experiment with techniques to improve the compile-time performance of the Chisel-generated C++ code, possibly with switches to optimize for compile-time or run-time.

10.8 FIFO

A generic FIFO could be defined as shown in Figure 3 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() };
```

Simulator	Compile Time (s)	Compile Speedup	Run Time (s)	Run Speedup	Total Time (s)	Total Speedup
VCS RTL simulator	22	1.000	5368	1.00	5390	1.00
Chisel C++ RTL simulator	119	0.184	575	9.33	694	7.77
Virtex-6 FPGA	3660	0.006	76	70.60	3736	1.44

Table 2: Comparison of simulation time between Chisel C++ simulator, Synopsys VCS Verilog simulation, and FPGA emulation, on a 64-bit five-stage RISC processor running an OS boot test.

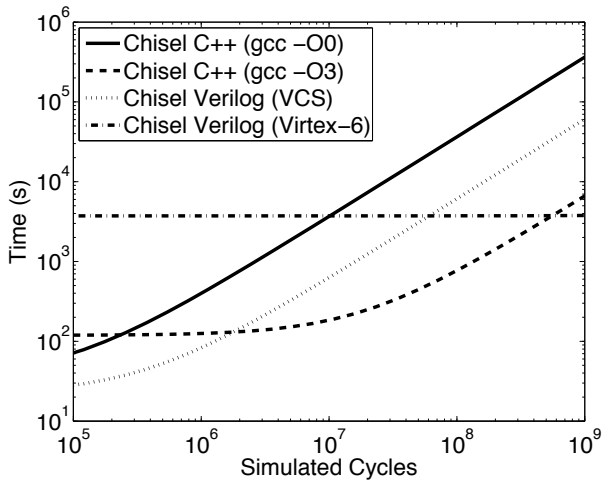

Figure 2: A comparison of total time required to compile and simulate a system using various backends from Chisel.

```
}
```

It is also possible to define a generic decoupled interface:

```
class ioDecoupled[T <: Data]()(data: => T) extends Bundle() {
  val ready = Bool(INPUT)
  val valid = Bool(OUTPUT)
  val bits  = data.asOutput
}
```

This template can then be used to add a handshaking protocol to any set of signals:

```
class decoupledDemo extends ioDecoupled()( new DataBundle() )
```

The FIFO interface in Figure 3 can be now be simplified as follows:

```
class FifoIO[T <: Data]()(gen: => T) extends Bundle()  {
  val enq = new ioDecoupled()( gen ).flip()
  val deq = new ioDecoupled()( gen )
}
```

10.9 Generated Verilog

Running the Chisel compiler on the FIFO example generates the Verilog code shown in Figure 4 .

The Verilog output from Chisel might need to be simulated together with other existing Verilog IP blocks. We compared the Verilog simulation speed of the Chisel-generated Verilog versus hand-written behavioral Verilog for a 64-bit data-parallel processor design, including pipelined single and double-precision FMA units, and a pipelined 64-bit integer multiplier. We ran 92 test assembly programs on both VCS-generated simulators. The Chisel-generated Verilog simulator was 1.65× slower in total than the behavioral Verilog simulator due to the low-level structural nature of the Verilog code generated by Chisel. However, we have not yet tuned the Verilog output for Verilog simulation performance, and we believe even the current slowdown is acceptable to enable co-simulation.

```
class FifoIO[T <: Data]()(gen: => T) extends Bundle() {
  val enq_val = Bool(INPUT)
  val enq_rdy = Bool(OUTPUT)
  val deq_val = Bool(OUTPUT)
  val deq_rdy = Bool(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: Parameterized FIFO example.

10.10 Chisel Components

Chisel has been in use for over a year and a number of components have been written in it. We developed the following components as part of our research infrastructure, many of which are used in the vector processor described in Section 10.11:

- clock dividers
- queues
- decoders, encoders, popcount
- scoreboards
- integer ALUs
- LFSR
- Booth multiplier, iterative divider
- ROMs, RAMs, CAMs
- TLB
- direct-mapped caches, set-associative blocking caches

```
module Fifo(input clk, input reset,
    input  io_enq_val,
    output io_enq_rdy,
    output io_deq_val,
    input  io_deq_rdy,
    input [31:0] io_enq_dat_A,
    input [31:0] io_enq_dat_B,
    output[31:0] io_deq_dat_A,
    output[31:0] io_deq_dat_B);

  wire T0;
  wire is_empty;
  wire T1;
  reg[4:0] deq_ptr;
  wire[4:0] T2;
  wire[4:0] deq_ptr_inc;
  wire do_deq;
  reg[4:0] enq_ptr;
  wire[4:0] T3;
  wire[4:0] enq_ptr_inc;
  wire do_enq;
  wire T4;
  reg[0:0] is_full;
  wire is_full_next;
  wire T5;
  wire T6;
  wire T7;
  wire T8;
  wire T9;
  wire T10;
  wire T11;

  assign io_deq_val = T0;
  assign T0 = ! is_empty;
  assign is_empty = T11 && T1;
  assign T1 = enq_ptr == deq_ptr;
  assign T2 = do_deq ? deq_ptr_inc : deq_ptr;
  assign deq_ptr_inc = deq_ptr + 1'h1/* 1*/;
  assign do_deq = io_deq_rdy && io_deq_val;
  assign T3 = do_enq ? enq_ptr_inc : enq_ptr;
  assign enq_ptr_inc = enq_ptr + 1'h1/* 1*/;
  assign do_enq = io_enq_rdy && io_enq_val;
  assign io_enq_rdy = T4;
  assign T4 = ! is_full;
  assign is_full_next = T7 ? 1'h1/* 1*/ : T5;
  assign T5 = T6 ? 1'h0/* 0*/ : is_full;
  assign T6 = do_deq && is_full;
  assign T7 = T9 && T8;
  assign T8 = enq_ptr_inc == deq_ptr;
  assign T9 = do_enq && T10;
  assign T10 = ~ do_deq;
  assign T11 = ! is_full;

  always @(posedge clk) begin
    if(reset) begin
      deq_ptr <= 5'h0/* 0*/;
    end else if(1'h1/* 1*/) begin
      deq_ptr <= T2;
    end
    if(reset) begin
      enq_ptr <= 5'h0/* 0*/;
    end else if(1'h1/* 1*/) begin
      enq_ptr <= T3;
    end
    if(reset) begin
      is_full <= 1'h0/* 0*/;
    end else if(1'h1/* 1*/) begin
      is_full <= is_full_next;
    end
  end
endmodule
```

Figure 4: Verilog Generated from Chisel for the FIFO example

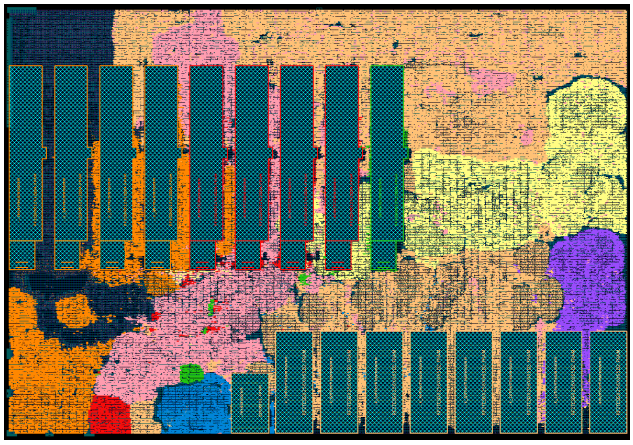

Figure 5: Data-parallel processor layout results

- direct-mapped caches, set-associative non-blocking caches
- prefetcher
- fixed-priority arbiters, round-robin arbiters
- single-precision, double-precision floating-point units
- 64-bit decoupled in-order single-issue 5-stage processor
- 64-bit vector unit (data-parallel processor)

We are working to factor these components into a standard library from which developers can more readily build large-scale designs.

We have taught a class in advanced computer architecture design where all students produced projects in Chisel. Example projects included accelerators for security, FFT, and spatial computing.

Additionally, Berkeley EECS graduate student Chris Celio is developing a number of educational processor microarchitectures with associated labs to help undergraduates learn computer architecture. These included a microcoded processor, one-stage, two-stage, and five-stage pipelines, and an out-of-order processor, all with accompanying visualizations.

10.11 Data-Parallel Processor Layout Results

The data-parallel processor layout results using IBM 45nm SOI 10-metal layer process using memory compiler generated 6T and 8T SRAM blocks are shown in Figure 5.