Dire Hardware Design Situation

- slow hardware design
  - 1980's style languages and baroque tool chains with ad hoc scripts
  - manual optimization obscuring designs
  - minimal compile-time and run-time errors
  - army of people in both CAD tools and design – costs $10Ms

- slow and expensive to synthesize
  - takes order days
  - not robust and so largely manual process
  - proprietary tools – cost > $1M / year / seat

- slow testing and evaluation
  - runs 200M x slower than code runs in production
  - army of verification people – costs $10Ms

- slow and expensive fabrication
  - very labor intensive – costs $1Ms

- design costs dominate
- very few ASIC designs and chip startups
“Iron Law of Design Performance”

\[ T_{design} = n \times (T_{program} + T_{build} + T_{eval}) \]

- every step in loop is potential bottleneck
- better results happen by iterating through design loop
- currently can only go around design loop a few times

- need to shorten design loop and get through it more times
- need to lower costs on all steps
Our Steps – “the Chisel Plan”

1. generation – use good software ideas – embedded host language
2. composition – design by composing bigger reusable pieces
3. transformation – specification + transformations = FIRRTL
4. optimization – parameterization + design space exploration
5. layering – incrementally higher level and strategic
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology

- will get increases in productivity and decreases in costs leading to
  - shorter time to market
  - more efficient designs
  - more “tapeouts” and chip startups

- UC Berkeley uniquely positioned to jump start revolution
  - strong VLSI tools and architecture programs
  - non-profit orientation and open source (BSD) tradition
(1) Hardware Generators

problem
- hardware design is too low level
- hard to write generators for family of designs

solution

*leverage great ideas in software engineering in hardware design*
- embed hardware construction in programming language
- leverage host language ideas and software engineering techniques
- zero cost abstractions
Status Quo Generators

1. write verilog design structurally – literal
2. verilog generate command – limited
3. write perl script that writes verilog – awkward

module counter (clk, reset);
    input clk;
    input reset;
    parameter W = 8;
    reg [W-1:0] cnt;
    always @ (posedge clk)
    begin
        if (reset)
            cnt <= 0
        else
            cnt <= cnt + 1
    end
endmodule
A hardware construction language
- “synthesizable by construction”
- creates graph representing hardware

Embedded within Scala language to leverage mindshare and language design

Best of hardware and software design ideas

Multiple targets
- Simulation and synthesis
- Memory IP is target-specific

Not Scala app -> Verilog arch
Well formed Chisel graphs are synthesizable.

- Use small number of basic nodes
  - simple semantics
  - easy to synthesize
- During construction check that
  - types, directions and widths match
  - there are no combinational loops

- If it passes these checks then it’s synthesizable
The Scala Programming Language

- Object Oriented
  - Factory Objects, Classes
  - Traits, overloading etc
  - Strongly typed with type inference

- Functional
  - Higher order functions
  - Anonymous functions
  - Currying etc

- Extensible
  - Domain Specific Languages (DSLs)

- Compiled to JVM
  - Good performance
  - Great Java interoperability
  - Mature debugging, execution environments

- Growing Popularity
  - Twitter
  - many Universities
- Chisel has 3 primitive datatypes
  - UInt – Unsigned integer
  - SInt – Signed integer
  - Bool – Boolean value
- Can do arithmetic and logic with these datatypes

**Example Literal Constructions**

```scala
val sel = Bool(false)
val a = UInt(25)
val b = SInt(-35)
```

where `val` is a Scala keyword used to declare variables whose values won’t change
Aggregate Data Types

Bundle

- User-extendable collection of values with named fields
- Similar to structs

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

Vec

- Create indexable collection of values
- Similar to array
- Useful Methods: contains, indexWhere, ...

```scala
val myVec = Vec.fill(5){ SInt(width=23) }
```
The user can construct new data types
  - Allows for compact, readable code

Example: Complex numbers
  - Useful for FFT, Correlator, other DSP
  - Define arithmetic on complex numbers

```scala
class Complex(val real: SInt, val imag: SInt) extends Bundle {
  def + (b: Complex): Complex =
    new Complex(real + b.real, imag + b.imag)
  ...
}
val a = new Complex(SInt(32), SInt(-16))
val b = new Complex(SInt(-15), SInt(21))
val c = a + b
```
class GCD extends Module {
    val io = new Bundle {
        val a = UInt(INPUT, 16)
        val b = UInt(INPUT, 16)
        val z = UInt(OUTPUT, 16)
        val valid = Bool(OUTPUT)
    }
    val x = Reg(init = io.a)
    val y = Reg(init = io.b)
    when (x > y) {
        x := x - y
    } .otherwise {
        y := y - x
    }
    io.z := x
    io.valid := y === UInt(0)
}
class Valid[T <: Data](dtype: T) extends Bundle {
    val data = dtype.clone
    val valid = Bool()
    override def clone = new Valid(dtype)
}

class GCD extends Module {
    val io = new Bundle {
        val a = UInt(INPUT, 16)
        val b = UInt(INPUT, 16)
        val out = new Valid(UInt(OUTPUT, 16))
    }
    ...  
    io.out.data := x
    io.out.valid := y === UInt(0)
}
abstract class Filter[T <: Data](dtype: T) extends Module {
    val io = new Bundle {
        val in  = new Valid(dtype).asInput
        val out = new Valid(dtype).asOutput
    }
}

class FunctionFilter[T <: Data](f: T => T, dtype: T) extends Filter(dtype) {
    io.out.valid := io.in.valid
    io.out    := f(io.in)
}
def clippingFilter[T <: Num](limit: Int, dtype: T) =
  new FunctionFilter(x => min(limit, max(-limit, x)), dtype)

def shiftingFilter[T <: Num](shift: Int, dtype: T) =
  new FunctionFilter(x => x >> shift, dtype)
class ChainedFilter[T <: Num](dtype: T) extends Filter(dtype) = {
  val shift = new ShiftFilter(2, dtype)
  val clipper = new ClippingFilter(1 << 7, dtype)
  io.in <> shift.io.in
  shift.io.out <> clipper.io.in
  clipper.io.out <> io.out
}
def delays[T <: Data](x: T, n: Int): List[T] = 
  if (n <= 1) List(x) else x :: delays(Reg(next = x), n-1)

def FIR[T <: Num](hs: Seq[T], x: T): T = 
  (hs, delays(x, hs.length)).zipped.map( _ * _ ).reduce( _ + _ )

class TstFIR extends Module {
  val io = new Bundle{ val x = SInt(INPUT, 8); val y = SInt(OUTPUT, 8) }
  val h = Array(SInt(1), SInt(2), SInt(4))
  io.y := FIR(h, io.x)
}

\[ y[n] = \sum_{k=0}^{N-1} x[n - k] h[k] \]
Other Generators

- Rocket Chip – cook + lee + waterman + ...
- BOOM – celio
- FFT generator – twigg
- Spectrometer – bailey
- Sha-3 generator – schmidt + izraelivitz + ...
- CS250 accelerators – berkeley EECS graduate students
- many more
Our Steps – “the Chisel Plan”

1. generation – use good software ideas – embedded host language
2. composition – design by composing bigger reusable pieces
3. transformation – specification + transformations = FIRRTL
4. optimization – parameterization + design space exploration
5. layering – incrementally higher level and strategic
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology
(2) Composition

problem

- people reinvent blocks over and over again
- how to reuse blocks and compose

solution

- build hardware out of bigger pieces
- construct common libraries and package manager
- build community
- provide target specific composers
open source on github accepting pull requests
website, mailing lists, blog, twitter
online documentation and tutorial
classes, bootcamps, and materials
library of high level and reusable components

> 1 FTE for community outreach, support, development

chisel.eecs.berkeley.edu
https://github.com/ucb-bar/chisel
Growing Library of Modules

- functional Vec interface,
- counters,
- shift-registers, pipes, queues,
- priority-mux, decoders, encoders,
- fixed-priority, round-robin, and locking arbiters,
- cross-bar, time multiplexor,
- popcount, scoreboards,
- ROMs, RAMs, CAMs, TLB, caches, prefetcher,
- integer ALUs, LFSR, Booth multiplier, iterative divider
- IEEE-754/2008 floating-point units
- processor + uncore building blocks
- Open Source Berkeley ISA
- Supports GCC, LLVM, Linux
- Accelerator Interface
- Growing Adoption – LowRisc etc
- Rocket-Chip is Processor Generator in Chisel

http://www.riscv.org
- NOC
- Memory Blocks
- IO / AXI interfaces

http://opensocfabric.lbl.gov/
Our Steps – “the Chisel Plan”

1. generation – use good software ideas – embedded host language
2. composition – design by composing bigger reusable pieces
3. transformation – specification + transformations = FIRRTL
4. optimization – parameterization + design space exploration
5. layering – incrementally higher level and strategic
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology
(3) Graph Transformations

**problem**
- designs are too complex and obfuscated with optimizations

**solution**
- factor design into
  - simple specification +
  - composable and reusable graph transformations
- standard RTL core called FIRRTL (virtually impossible in verilog)
  - file formats and API
  - language neutral

programmatic insertion of activity counters on registers
- clean intermediate representation (IR) for code
  - defined API
  - file format
  - tools
- easier to write
  - front-ends
  - transformational passes
  - back-ends
- leads to explosion in
  - languages
  - compilers
  - architectures
Flexible Intermediate Representation for RTL (FIRRTL)

- language neutral RTL IR with text format
- “LLVM for hardware”
- simple core
- semantic / structural information
- annotations
Example Passes

- rate balancing
  - auto pipelining
  - auto threading
  - unrolling
- evaluation and debug
  - activity counters
  - snapshot dumping / restoring
- additional features
  - fault tolerance
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>generation – use good software ideas – embedded host language</td>
</tr>
<tr>
<td>2</td>
<td>composition – design by composing bigger reusable pieces</td>
</tr>
<tr>
<td>3</td>
<td>transformation – specification + transformations = FIRRTL</td>
</tr>
<tr>
<td>4</td>
<td>optimization – parameterization + design space exploration</td>
</tr>
<tr>
<td>5</td>
<td>layering – incrementally higher level and strategic</td>
</tr>
<tr>
<td>6</td>
<td>simulation – speed up testing and evaluation methods</td>
</tr>
<tr>
<td>7</td>
<td>realization – fast + affordable deployment technology</td>
</tr>
</tbody>
</table>
problem
- hard to find great designs by hand

solution: facility for parameterizing and searching design space
- called Jackhammer – autotuner for hardware
- framework for organizing parameters
- language for specifying objective function
- parallel mechanism for optimizing over design space
simple solution doesn’t work:

```scala
class Cache(lineSize: Int, ...) extends Module ...
```

need first class parameters

- organize parameters and thread through construction
- want to split specification from hierarchy from exploration
- designs are hierarchical and need to specify various elements
- what’s minimal description that is robust to changes in hierarchy?
- how to constrain parameters?
- how to export design space?

got a solution in Chisel ...
- tuning parameters on rocket-chip on workloads
- ability to launch thousands of results on cluster
- present pareto optimal plots
Our Steps – “the Chisel Plan”

1. generation – use good software ideas – embedded host language
2. composition – design by composing bigger reusable pieces
3. transformation – specification + transformations = FIRRTL
4. optimization – parameterization + design space exploration
5. layering – incrementally higher level and strategic
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology
problem

- HLS is intractable
- no dominant design language (silver bullet)

solution:

- series of layered languages
- mix and match strategies
- zero or measurable overhead
Our Steps – “the Chisel Plan”

1. generation – use good software ideas – embedded host language
2. composition – design by composing bigger reusable pieces
3. transformation – specification + transformations = FIRRTL
4. optimization – parameterization + design space exploration
5. layered languages – incrementally higher level and strategic
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology
problem

- simulation/evaluation is a big bottleneck
- choose fast runtime or fast compile time
- difficult to debug hardware

solution

- pay as you go emulation with DREAMER
- statistical power estimation
- cycle accurate multicore on multicore

program ➔ build
eval ➔ fab
C++ Simulator

- cycle accurate simulator
  - easy way to debug designs
- compiles Chisel to one C++ class
  - expand multi words into single word operations
  - topologically sorts nodes based on dependencies
- simulates using two phases
  - `clock_lo` for combinational
  - `clock_hi` for state updates
Comparison of simulation time when booting Tessellation OS

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Compile Time (s)</th>
<th>Compile Speedup</th>
<th>Run Time (s)</th>
<th>Run Speedup</th>
<th>Total Time (s)</th>
<th>Total Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCS</td>
<td>22</td>
<td>1.000</td>
<td>5368</td>
<td>1.00</td>
<td>5390</td>
<td>1.00</td>
</tr>
<tr>
<td>Chisel C++</td>
<td>119</td>
<td>0.184</td>
<td>575</td>
<td>9.33</td>
<td>694</td>
<td>7.77</td>
</tr>
<tr>
<td>Virtex-6</td>
<td>3660</td>
<td>0.006</td>
<td>76</td>
<td>70.60</td>
<td>3736</td>
<td>1.44</td>
</tr>
</tbody>
</table>
Simulation Crossover Points

Time to Cycle N for Emulating

- c
- fpga
- work in progress...
- OpenNOC pushing our C++ backend tools
- scalable compile time and run time performance
- automatically split graph into combinational islands
- can split into separate functions/files
- parallelize compilation and execution
- already have much better compile times
- x86 barrier runs at a 1MHz
- could scale up multicore cycle accurate simulation
- standard protocol text based protocol
- peek poke step snapshot ...
scala interface to debug interface using chisel names

advanced tester allows decoupled support
Our Steps – “the Chisel Plan”

1. generation – use good software ideas – embedded host language
2. composition – design by composing bigger reusable pieces
3. transformation – specification + transformations = FIRRTL
4. optimization – parameterization + design space exploration
5. layered languages – incrementally higher level and strategic
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology
(7) Create Affordable Deployment Technology

problem
- fabbing ASICs is expensive and time consuming
- FPGAs are hard to program and use

solution
- new coarse grained “FPGA”* based on DREAMER
- fast / open source tools
- 3-stage RISCV CPU hand-coded in Verilog
- Translated to Chisel
- Resulted in 3x reduction in lines of code
- Most savings in wiring
- Lots more savings to go ...
UC Berkeley Classes

- 3x CS152 – Undergraduate Computer Architecture
  - Sodor edu cores – 1/2/3/5 stage, microcode, out of order, ...
  - Multicore and Vector

- 3x CS250 – VLSI System Design
  - Processors
  - Image Processing
  - RISC-V Rocket Accelerators

- 1x CS294-88 – Declarative Design Seminar
  - High Level Specification
  - Automated Design Space Exploration

- 1x EE290C – Advanced Topics in Circuits
  - DSP ASICS
  - Software Defined Radio
Chisel Projects

- SOC and NOC generator – Lawrence Berkeley National Labs
- NOC generator – Microsoft Research
- Oblivious RAM – Maas et al @ Berkeley
- Garbage Collector – Maas et al @ Berkeley
- Out of Order Processor Generator – Celio et al @ Berkeley
- Spectrometer Chip – Nasa JPL / Berkeley
- Monte Carlo Simulator – TU Kaiserslautern
- Precision Timed Machine (Patmos) – EC funded project
- Precision Timed Machine (PRET) – Edward Lee’s Group
- Chisel-Q – Quantum Backend – John Kubiatowicz’s Group
- Gorilla++ – Abstract Actor Network Language
- LowRisc – New Raspberry Pi SOC
- RiscV Processors and Uncore – Madras IIT
- Evidence of many other projects on mailing lists
<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADTs</td>
</tr>
<tr>
<td>DSLs</td>
</tr>
<tr>
<td>FP</td>
</tr>
<tr>
<td>OOP</td>
</tr>
<tr>
<td>GAA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Verilog</th>
<th>SystemVerilog</th>
<th>Bluespec</th>
<th>Chisel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADTs</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DSLs</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>FP</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OOP</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>GAA</td>
<td>no</td>
<td>no</td>
<td>only</td>
<td>yes*</td>
</tr>
<tr>
<td>open source</td>
<td>yes**</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

where **ADT** is Abstract Data Types, **DSL** is Domain Specific Language, **FP** is Functional Programming, **OOP** is Object Oriented Programming, and **GAA** is Guarded Atomic Actions.

* can layer it on top of Chisel

** although simulator free, rest of tools still cost money
Conclusions

Advocated a six step plan

1. generation – use best software engineering ideas
2. composition – bigger pieces and network effects
3. transformation – spec + transformations FIRRTL
4. layers – incrementally higher level and focussed
5. optimization – design space exploration
6. simulation – speed up testing and evaluation methods
7. realization – fast + affordable deployment technology

Note that

- better RTL design is already a win
- chisel library and community are growing
- there are huge opportunities for improving design cycle
- there are lots of low hanging fruit along the way
Current But Lesser Known Features

- fix point types
- floating point types
- complex numbers
- parameterization
- jackhammer
- multiple clock domains
- OpenNOC – http://opensocfabric.lbl.gov/
- debug api
- chisel random tester
- FIRRTL draft spec ASAP
Features Available in Next 3-6 Months

- basics
  - llvm backend

- composition
  - software defined platforms
  - arduino for fpga’s
  - processor toolkit
  - advanced cookbook
  - ROCC accelerators
  - standard library
  - library mechanism
  - advanced cookbook

- spec + transformation
  - FIRRTL implementation
  - Chisel 3.0 + FIRRTL
  - tranformations
    - scan chains
    - auto pipelining
    - ...

- declarative
  - DSE DSL

- layered languages
  - streaming DSP DSL
  - transactor DSL

- simulation
  - chisualizer
  - debug machine on fpga’s
  - statistical power sampling
  - multicore on multicore simulation
- Website – chisel.eecs.berkeley.edu
- Getting Started Guide – documentation
- RoCC Labs – Next
- Sodor Cores – https://github.com/ucb-bar/riscv-sodor/
- Rocket Chip Generator – www.riscv.org
- Bootcamp at HPCA – http://darksilicon.org/hpca
- Chisel Sources – https://github.com/ucb-bar/chisel/
Workshop / Bootcamp
HPCA – IEEE High Performance Computer Architecture
All Day Saturday Feb 7th
San Francisco Airport Marriott Waterfront Hotel
Register – http://darksilicon.org/hpca
Sign up for chisel tutorial on saturday
git clone https://github.com/ucb-bar/chisel-tutorial.git

cd chisel-tutorial
https://chisel.eecs.berkeley.edu/documentation.html

getting started  getting-started.pdf
tutorial        tutorial.pdf
manual          manual.pdf

https://github.com/ucb-bar/chisel/

setup          readme.md
utils          src/main/scala/ChiselUtils.scala

https://chisel.eecs.berkeley.edu/download.html

sodor          https://github.com/ucb-bar/riscv-sodor/
website  chisel.eecs.berkeley.edu
mailing list  groups.google.com/group/chisel-users
github  https://github.com/ucb-bar/chisel/
features + bugs  https://github.com/ucb-bar/chisel/issues
more questions  stackoverflow.com/questions/tagged/chisel
twitter  #chiselhdl
me  jrb@eecs.berkeley.edu
Funding

funding initiated under

- Project Isis: under DoE Award DE-SC0003624.
- Par Lab: Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.

ongoing support from

- ASPIRE: DARPA PERFECT program, Award HR0011-12-2-0016. Additional support comes from Aspire affiliates Google, Intel, Nokia, Nvidia, Oracle, and Samsung.
- CAL (with LBNL): under Contract No. DE-AC02-05CH11231.