

Chisel: Constructing Hardware In a Scala Embedded Language

Jonathan Bachrach, Huy Vo, Brian Richards,
Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook,
John Wawrzynek, Krste Asanovic

EECS UC Berkeley

April 21, 2022

Harder to get hardware / software efficiency gains

- Need massive design-space exploration
 - Hardware and software codesign and cotuning
- Need meaningful results
 - Cycle counts
 - Cycle time, power and area
 - Real chips
- Traditional architectural simulators, hardware-description languages, and tools are inadequate
 - Slow
 - Inaccurate
 - Error prone
 - Difficult to modify and parameterize

Make it

- Easier to make design changes
 - Fewer lines of design code (» **3x**)
 - More reusable code
 - Parameterize designs
- Faster to test results (» **8x**)
 - Fast compilation
 - Fast simulation
 - Easy testing
 - Easy verification

Result

- Explore more design space

- Best of hardware and software design ideas
- Embedded within Scala language to leverage mindshare and language design
- Algebraic construction and wiring
- Hierarchical, object oriented, and functional construction
- Abstract data types and interfaces
- Bulk connections
- Multiple targets
 - Simulation and synthesis
 - Memory IP is target-specific

single source

../manual/figs/targets.pdf

The diagram consists of a large rectangular box. Above the box is the text 'single source'. Inside the box, centered, is the text ' ../manual/figs/targets.pdf '. Below the box is the text 'multiple targets'.

multiple targets

- Compiled to JVM
 - Good performance
 - Great Java interoperability
 - Mature debugging, execution environments
- Object Oriented
 - Factory Objects, Classes
 - Traits, overloading etc
- Functional
 - Higher order functions
 - Anonymous functions
 - Currying etc
- Extensible
 - Domain Specific Languages (DSLs)

figs/programming-

figs/programming-

$\text{Mux}(x > y, x, y)$

[figs/max2.pdf](#)

```
class Max2 extends Component {  
  val io = new Bundle {  
    val x = UFix(width = 8).asInput  
    val y = UFix(width = 8).asInput  
    val z = UFix(width = 8).asOutput }  
  io.z := Mux(io.x > io.y, io.x, io.y)  
}
```

figs/Max2c.pdf

```
val m1 = new Max2()  
m1.io.x := a  
m1.io.y := b  
val m2 = new Max2()  
m2.io.x := c  
m2.io.y := d  
val m3 = new Max2()  
m3.io.x := m1.io.z  
m3.io.y := m2.io.z
```

figs/Max4.pdf


```
def Max2 = Mux(x > y, x, y)
```

```
Max2(x, y)
```

figs/Max2.pdf

```
Reduce(Array(a, b, c, d), Max2)
```

[figs/reduceMax.pdf](#)

```
class GCD extends Component {  
  val io = new Bundle {  
    val a      = UFix(INPUT, 16)  
    val b      = UFix(INPUT, 16)  
    val z      = UFix(OUTPUT, 16)  
    val valid  = Bool(OUTPUT) }  
  val x = Reg(resetVal = io.a)  
  val y = Reg(resetVal = io.b)  
  when (x > y) {  
    x := x - y  
  } .otherwise {  
    y := y - x  
  }  
  io.z      := x  
  io.valid := y === UFix(0)  
}
```

figs/gcd.pdf

- Chisel has 4 primitive datatypes
 - `Bits` – raw collection of bits
 - `Fix` – signed fixed-point number
 - `UFix` – unsigned fixed-point number
 - `Bool` – Boolean value
- Can do arithmetic and logic with these datatypes

Example Literal Constructions

```
val sel = Bool(false)
val a   = UFix(25)
val b   = Fix(-35)
```

where `val` is a Scala keyword used to declare variables whose values won't change

Bundle

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

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

Vec

- Create indexable collection of values
- Similar to arrays

```
val myVec = Vec(5){ Fix(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

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

val a = new Complex(Fix(32), Fix(-16))
val b = new Complex(Fix(-15), Fix(21))
val c = a + b
```

- Chisel users can define their own parameterized functions
 - Parameterization encourages reusability
 - Data types can be inferred and propagated

Example Shift Register:

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

where

- The input x is delayed n cycles
- x can be of any type that extends from `Data`

`Chain(n, in, x => f(x))`

`Map(ins, x => x * y)`

`figs/map.pdf`

`figs/chain.pdf`

`Reduce(data, Max)`


```
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 = log2(cache_depth).toInt
  val addr_off_width = log2(line_size/32).toInt
  val addr_tag_width = 32 - addr_idx_width - addr_off_width - 2
  val log2_assoc      = log2(associativity).toInt
  ...
  if (cache_type == DIR_MAPPED)
    ...
}
```

Simplest element is positive edge triggered register:

```
val prev_in = Reg(in)
```

Can assign data input later using wiring

```
val pc = Reg(){ UFix(width = 16) }  
pc := pc + UFix(1, 16)
```

Can quickly define more useful circuits

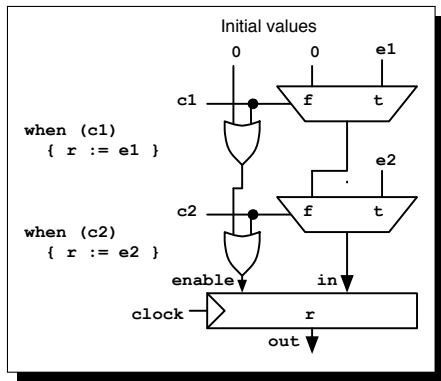
```
def risingEdge(x: Bool) = x && !Reg(x)
```

Convenient to specify updates
spread across several
statements

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

or

```
when (c1) { r := e1 }  
when (c2) { r := e2 }
```



Nesting

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

Chaining

```
when (c1) { u1 }  
.elsewhen (c2) { u2 }  
.otherwise { ud }
```

Dynamic Scoping

```
def condUpdateR (c: Bool, d: Data) = when (c) { r := d }
```

```
when (a) { condUpdateR(b, x) }
```

```
when (a) { when (b) { r := x } }
```

Regs and Wires

```
x := init
when (isEnabled) {
  x := data
}
```

Vecs and Mems

```
when (isEnabled) {
  m(addr) := data
}
```

```
val in  = (new DeqIo()){ new Packet() }
val out = (new EnqIo()){ new Packet() }
when (in.valid && out.ready) {
    out.enq(filter(in.deq()))
}
```

figs/filter.pdf

```
val in  = (new DeqIo()){ new Packet() }
val outs = Vec(4){ new EnqIo(){ new Packet() } }
val tbl = Mem(4){ UFix(width = 2) }
when (in.valid) {
    val k = tbl(in.data.header)
    when (outs(k).ready) {
        outs(k).enq(in.deq())
    }
}
```

```
class Mux2IO extends Bundle {  
  val sel = Bits(width = 1).asInput  
  val in0 = Bits(width = 1).asInput  
  val in1 = Bits(width = 1).asInput  
  val out = Bits(width = 1).asOutput  
}
```

figs/mux2.pdf

```
class Mux2Tests extends Iterator[Mux2IO] {  
  var i = 0  
  val n = pow(2, 3)  
  def hasNext = i < n  
  def next = {  
    val io = new Mux2IO  
    val k = Bits(i, width = log2up(n))  
    io.sel := k(0)  
    io.in0 := k(1)  
    io.in1 := k(2)  
    io.out := Mux(k(0), k(1), k(2))  
    i += 1  
    io  
  }  
}
```

- ~5200 lines total
- Embeds into Scala well

figs/linecount.png

- 3-stage RISC-V 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 ...

Composeable State Machines

```
Do{ ... }  
Exec(c){ a } / Exec{ a }  
Stop  
Skip / Wait(n)  
Seq(a, ...)  
Par(a, ...)  
Alt(c, a1, a2)  
While(c){ a } / Loop{ a }
```

Each process block uses a when

```
when (io.start) { ... }
```

to ensure that state updates are
updated only when process execute.

[figs/process.pdf](#)

```

class Multiply extends Component {
  val io = new Bundle{
    val start = Bool(INPUT);
    val x      = UFix(dir = INPUT, width = 32)
    val y      = UFix(dir = INPUT, width = 32)
    val z      = UFix(dir = OUTPUT, width = 32)
    val finish = Bool(OUTPUT) }
  val a  = Reg(){ UFix(0, 32) }
  val b  = Reg(){ UFix(0, 32) }
  val acc = Reg(){ UFix(0, 32) }
  val finish =
    Exec(io.start) {
      Seq(Do{ a := io.x; b := io.y; acc := UFix(0, 32) },
        While(b != UFix(0, 32)) {
          Do{ a := (a << UFix(1))
              b := (b >> UFix(1))
              acc := Mux(b(0) == Bits(1), acc+a, acc) } })
    }
  io.finish := finish
  io.z      := acc
}

```

```
class Router extends Transactor {  
  val n    = 2  
  val io   = new RouterIO(n)  
  val tbl  = Mem(32){ UFix(width = sizeof(n)) }  
  defRule("rd") {  
    val cmd = io.reads.deq()  
    io.replies.enq(tbl.read(cmd.addr))  
  }  
  defRule("wr") {  
    val cmd = io.writes.deq()  
    tbl.write(cmd.addr, cmd.data)  
  }  
  defRule("rt") {  
    val pkt = io.in.deq()  
    io.outs(tbl.read(pkt.header)).enq(pkt)  
  }  
}
```

figs/trouter.pdf

- 6-stage RISC decoupled integer datapath + 5-stage IEEE FPU + MMU and non-blocking caches
- Completely written in Chisel

`figs/rocket-microarchitecture.pdf`

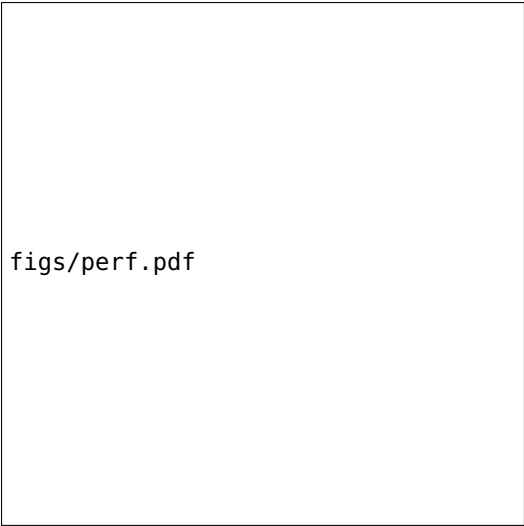
single source

```
../manual/figs/targets.pdf
```

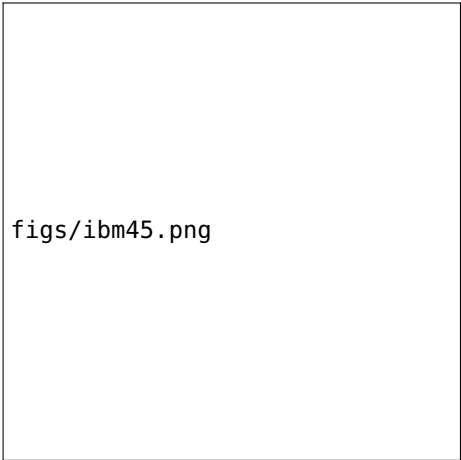
- Compiles to single class
 - Keep state and top level io in class fields
 - `clock_lo` and `clock_hi` methods
- Generates calls to fast multiword library using C++ templates
 - specializing for small word cases
 - remove branching as much as possible to utilize maximum ILP in processor

Comparison of simulation time when booting Tessellation OS

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



figs/perf.pdf



figs/ibm45.png

Completely written in Chisel

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

- Open source with BSD license
 - `chisel.eecs.berkeley.edu`
 - complete set of documentation
 - bootcamp / release june 8, 2012
- Library of components
 - queues, decoders, encoders, popcount, scoreboards, integer ALUs, LFSR, Booth multiplier, iterative divider, ROMs, RAMs, CAMs, TLB, caches, prefetcher, fixed-priority arbiters, round-robin arbiters, IEEE-754/2008 floating-point units
- Set of educational processors including:
 - microcoded processor, one-stage, two-stage, and five-stage pipelines, and an out-of-order processor, all with accompanying visualizations.

- Automated design space exploration
- Insertion of activity counters for power monitors
- Automatic fault insertion
- Faster and more scalable simulation
- More generators
- More little languages
- Compilation to UCLID