ASPIRE
UC BERKELEY

Chisel 2.0.0 to Chisel 3.0.0

Because everybody loves a trilogy

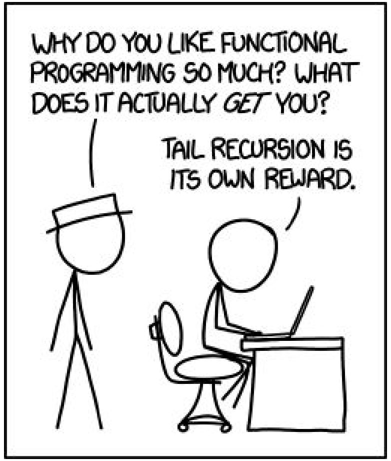
Presentation by Adam Izraelevitz

A long time ago, in a laboratory (not) far, far away....

Generators: Type-Safe Meta-Programming for RTL

Design Reuse

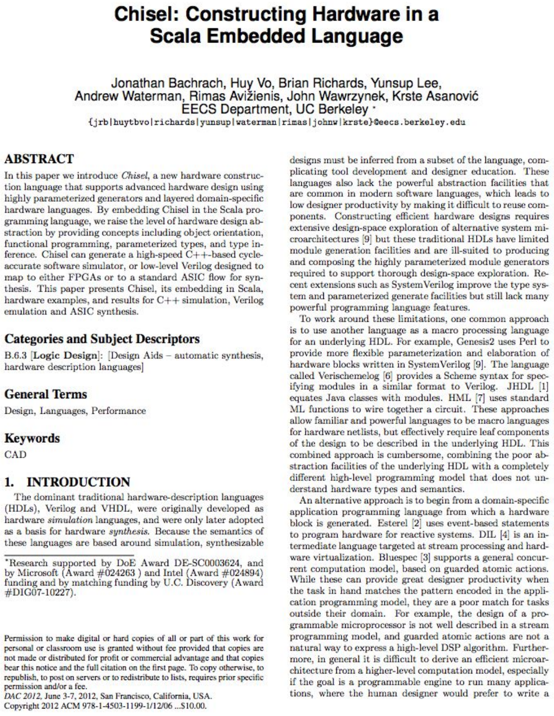
Type-Safety



WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

Powerful Language Features

Hired Jonathan Bachrach



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.

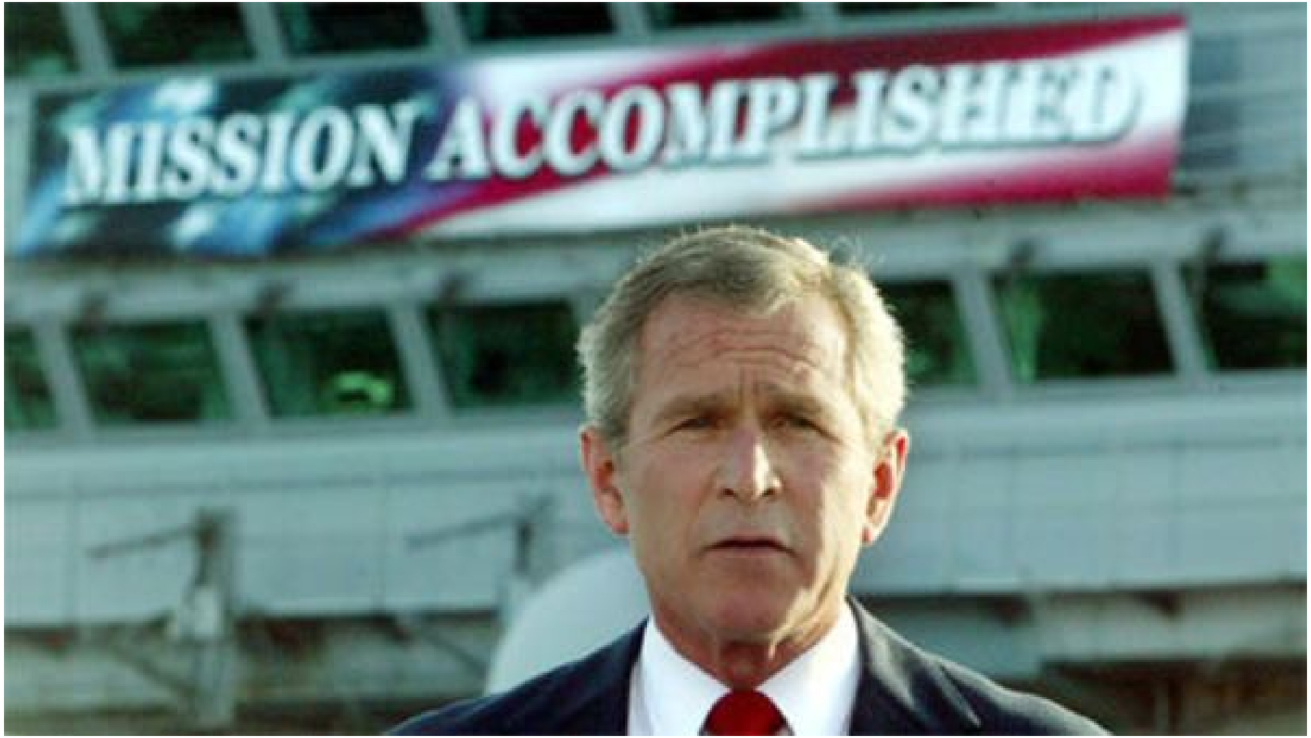
An alternative approach is to begin from a domain-specific application programming language from which a hardware block is generated. Esterel [2] uses event-based statements to program hardware for reactive systems. DIL [4] is an intermediate language targeted at stream processing and hardware virtualization. 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.

At the end of ParLab, we solved hardware design

ParLab Chip Highlight Reel


MISSION ACCOMPLISHED

And so, the problem of hardware design was forever solved.

Just kidding.

Solving Hardware Design ≠ Solving The Hardware Design Loop

* from “How to Draw Chip and Dale booklet”. (Walt Disney, 1955)

What had to change?

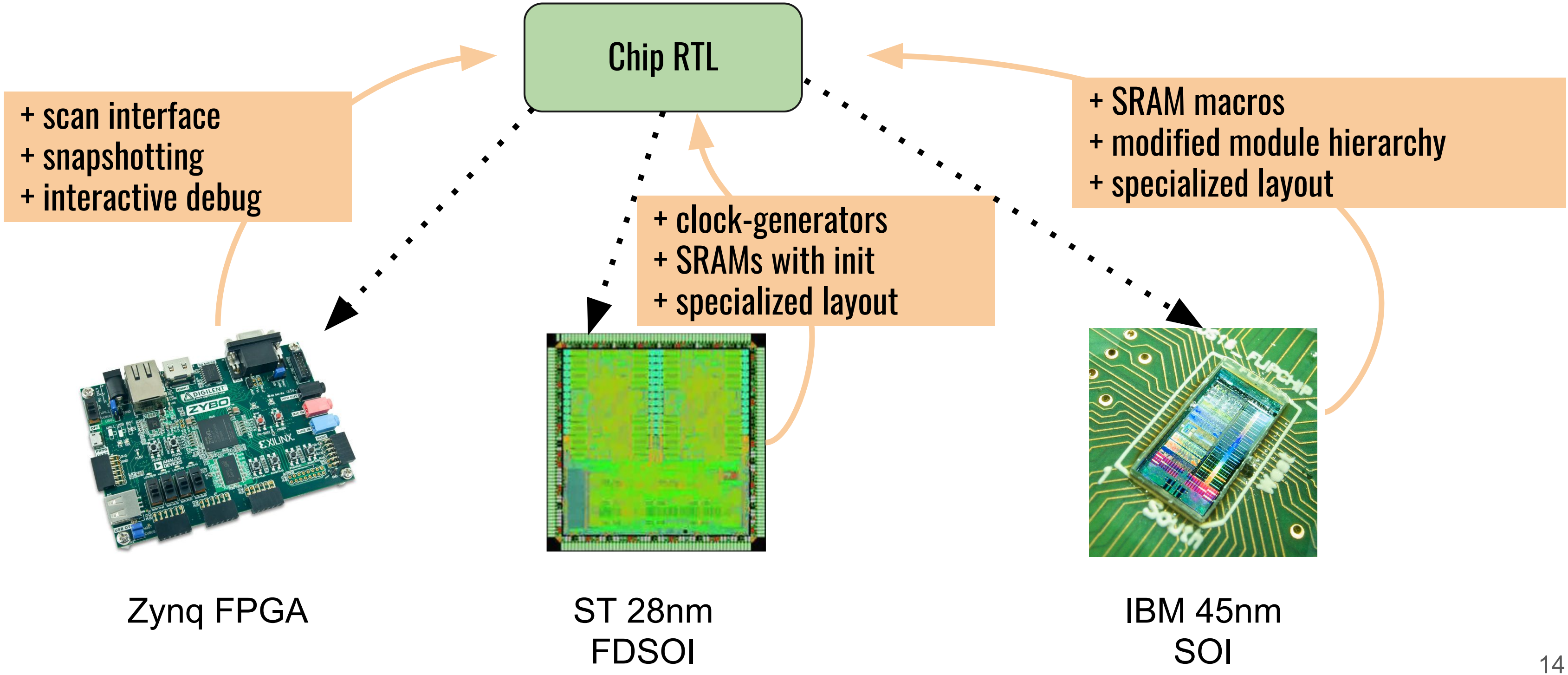
Hardware Design Ecosystem

CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

Stable and User-Friendly API

External Collaborations

Platform-Specific or Application-Specific RTL Changes



Chip RTL

+ scan interface
+ snapshotting
+ interactive debug

+ clock-generators
+ SRAMs with init
+ specialized layout

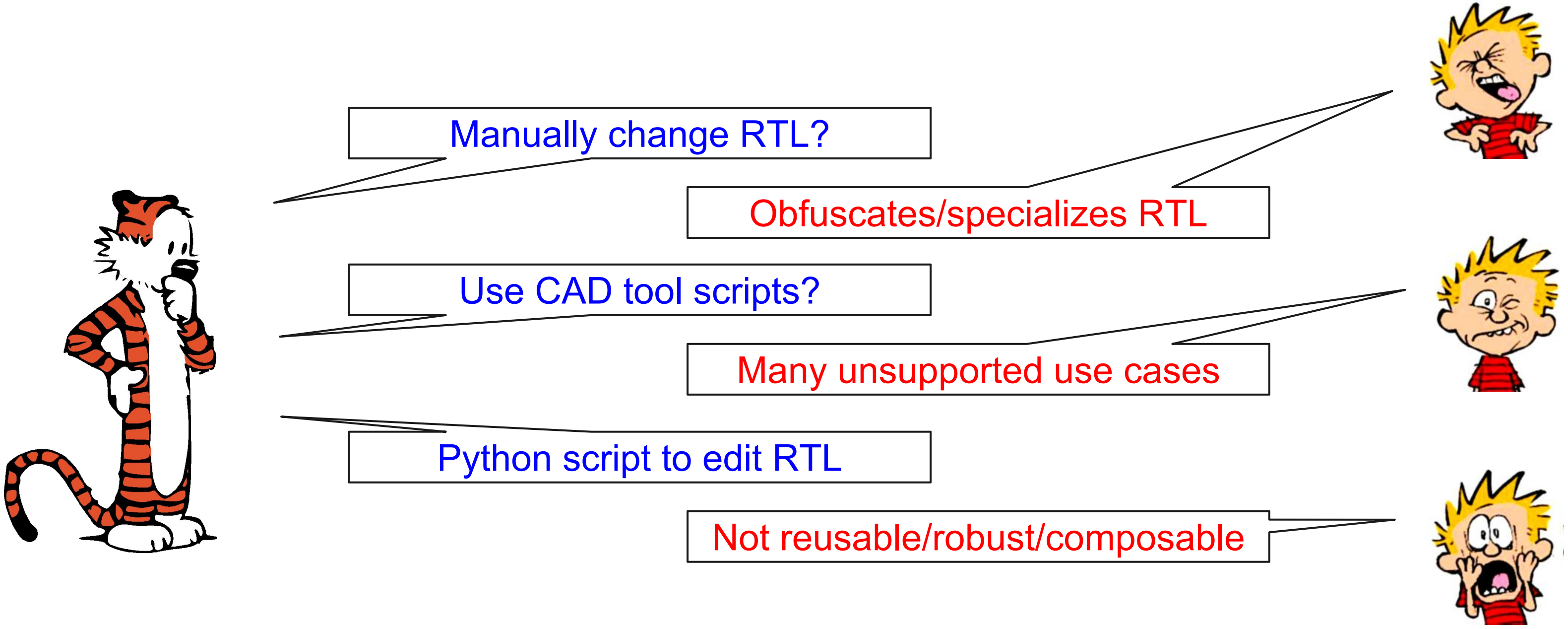
+ SRAM macros
+ modified module hierarchy
+ specialized layout

Zynq FPGA

ST 28nm
FDSOI

IBM 45nm
SOI

What were we doing?



Manually change RTL?

Obfuscates/specializes RTL

Use CAD tool scripts?

Many unsupported use cases

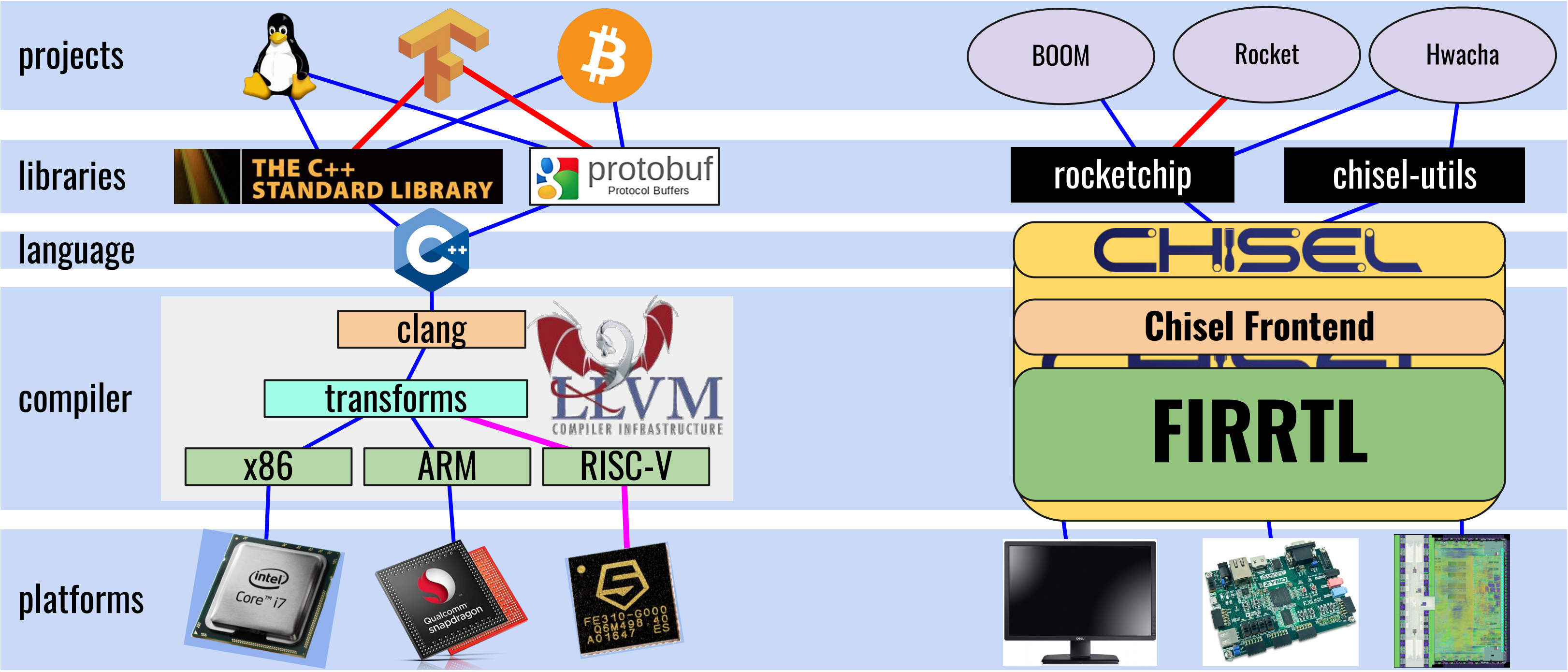
Python script to edit RTL

Not reusable/robust/composable

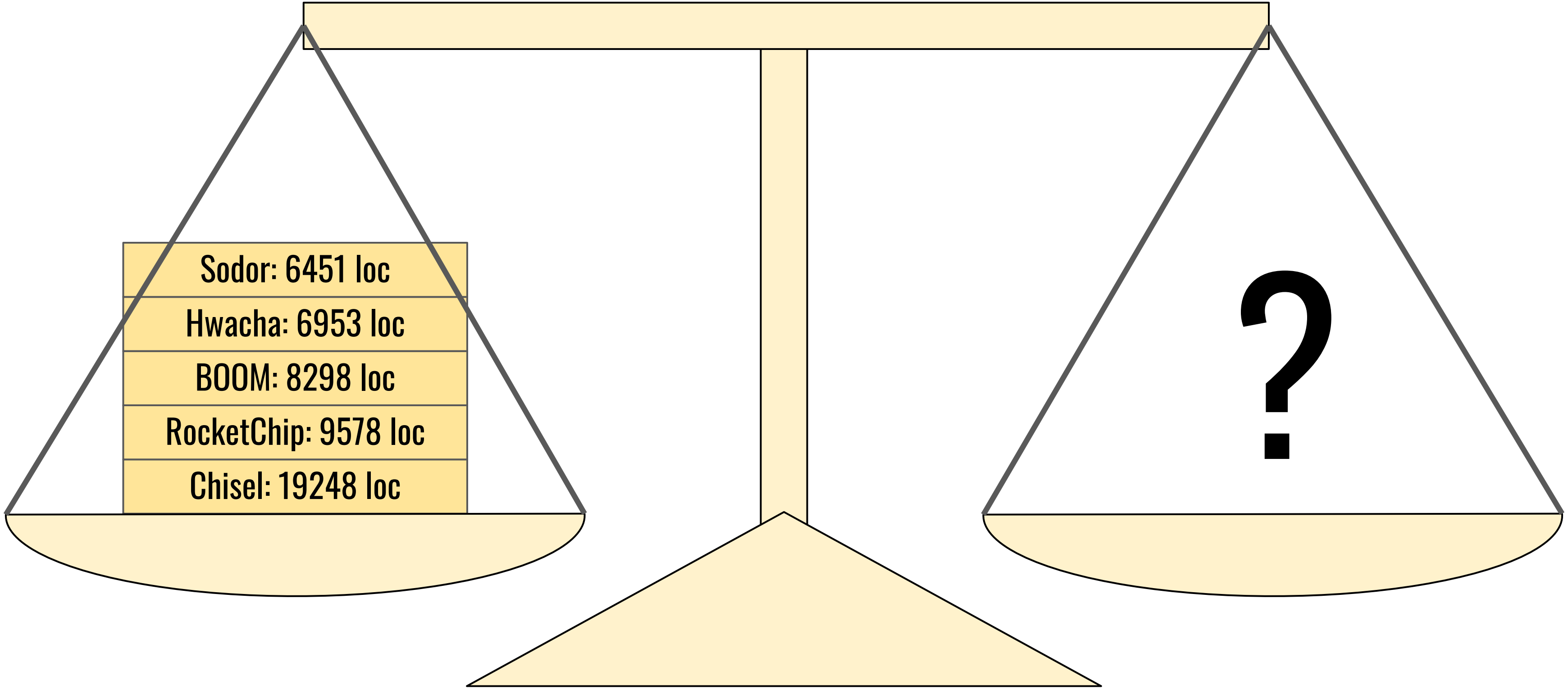
* from http://www.zoom-comics.com/wp-content/uploads/sites/36/2011/01/calvin-and-hobbes-faces.jpg, accessed 10/23/17

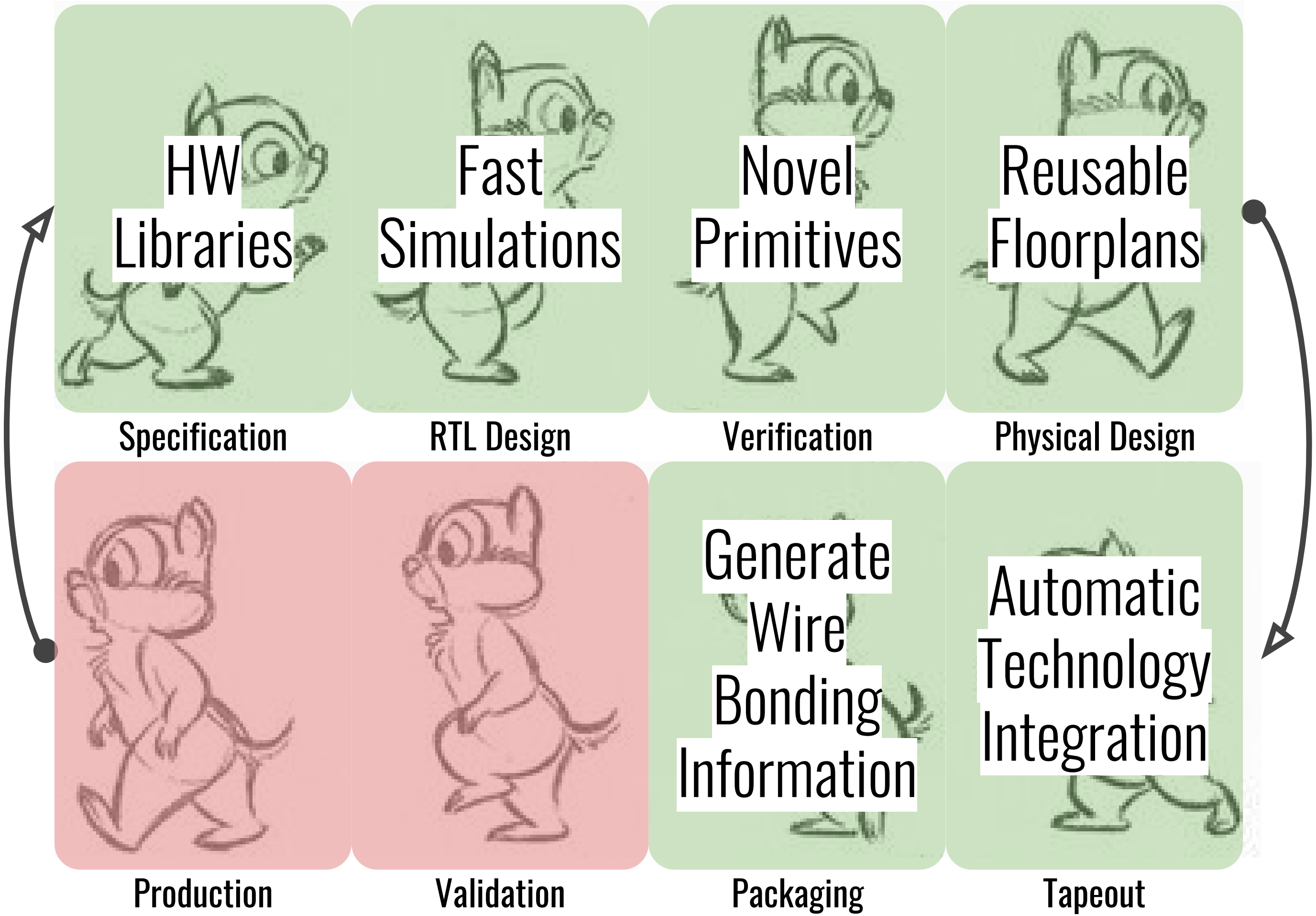
* from https://useravatar.services.amuniversal.com/user_avatars/avatars_gocomicsver3/3001000/3001618/IMG_7625.GIF, accessed 10/23/17

Realization: We need a software stack, but for hardware



Second-System-Syndrome: But... do we *really* need all that?





HW Libraries

Specification

Fast Simulations

RTL Design

Novel Primitives

Verification

Reusable Floorplans

Physical Design

Production

Validation

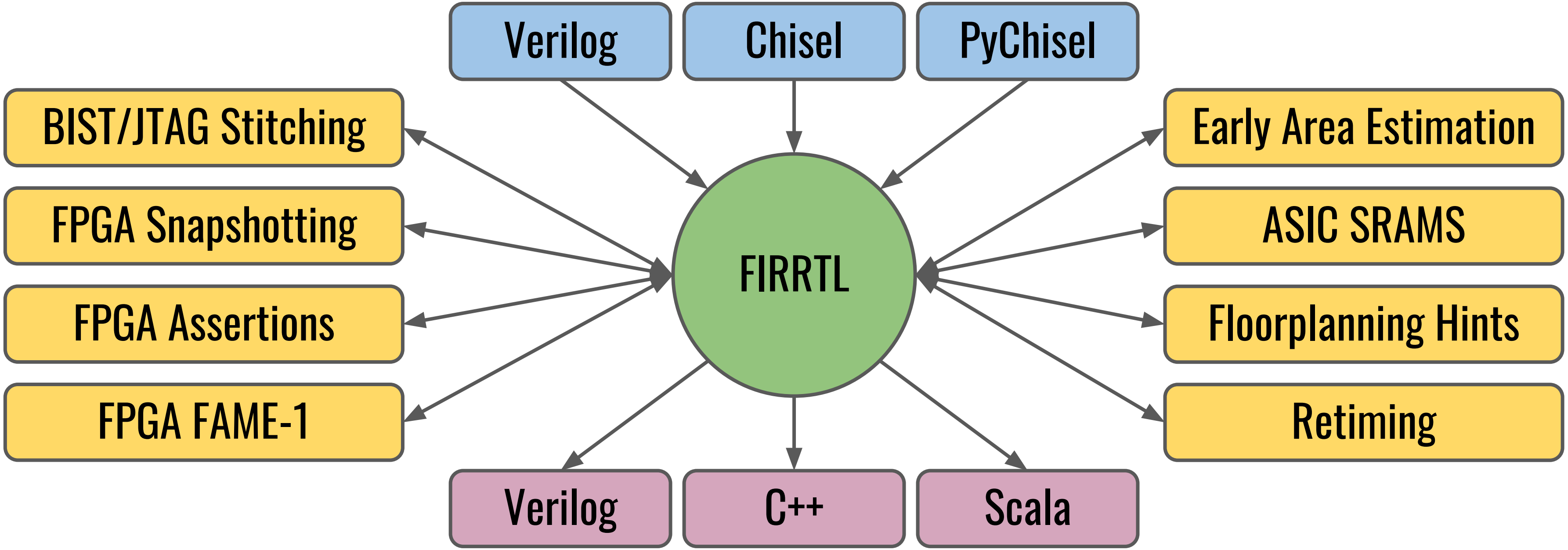
Generate Wire Bonding Information

Packaging

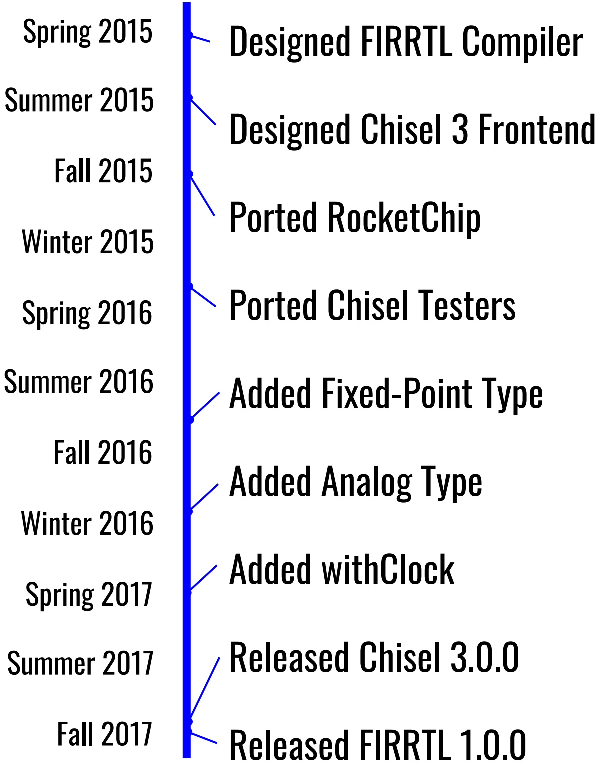
Automatic Technology Integration

Tapeout

For impact, we need an ecosystem


Verilog
Chisel
PyChisel
BIST/JTAG Stitching
FPGA Snapshotting
FPGA Assertions
FPGA FAME-1
FIRRTL
Early Area Estimation
ASIC SRAMS
Floorplanning Hints
Retiming
Verilog
C++
Scala

Developing, Porting, Code Reviews, Testing, and so forth



Andrew Waterman <waterman@cs.berkeley.edu>
Yunsup Lee <yunsup@sifive.com>
Adam Izraelevitz <izraelevitz@google.com>
jackkoenig <jack.koenig3@eecs.berkeley.edu>

Chisel 3.0.0 and FIRRTL 1.0.0 have been released!

- Projects
  - FireSim - Datacenter Emulation on FPGAs
  - Strober - Fast+Accurate Power Sims. for Long Programs
  - Hurricane 2 - Multi-Core DVFS (Sub-Core)
- Transformations
  - Quick (and Semi-Accurate) Timing and Area Estimation
  - Automatic Combinational Cycle Removal
  - Snapshotting and Hardware Assertions for FPGAs
- New Features
  - Hardware Types vs Hardware Components
  - New types (e.g. Complex, DspReal, Fixed-Point, Analog)
  - Chisel Library support (annotations)
  - Invalidate API for safer connections

Chisel 3

https://github.com/freechipsproject/chisel3/releases/tag/v3.0.0

Latest release

v3.0.0

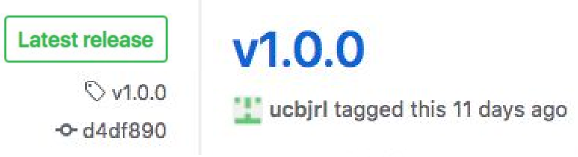
78bfa07

v3.0.0

ucbjrl released this 9 days ago

FIRRTL

https://github.com/freechipsproject/firrtl/releases/tag/v1.0.0



Latest release

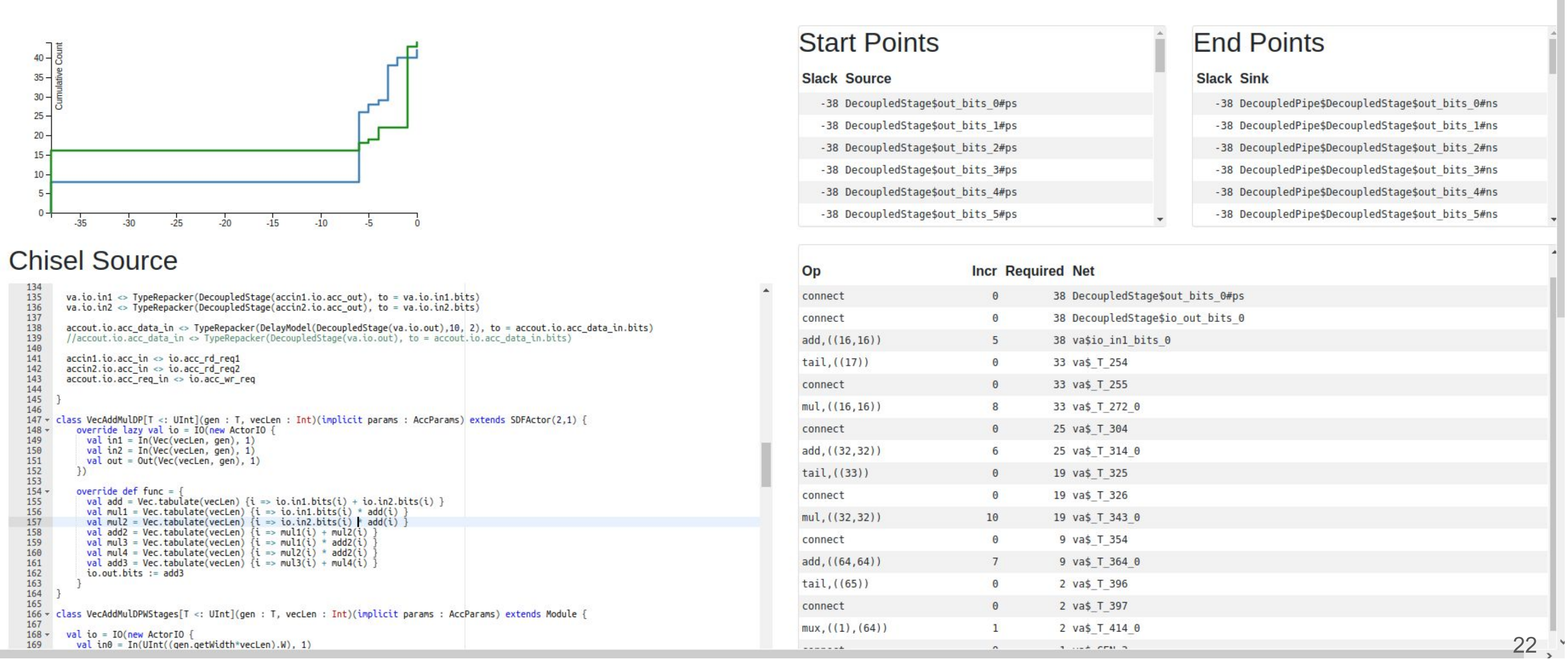
v1.0.0

d4df890

v1.0.0

ucbjrl tagged this 11 days ago

Intel: Fast and Semi-Accurate FIRRTL Timer



Start Points

Slack	Source
-38	DecoupledStage$out_bits_0#ps
-38	DecoupledStage$out_bits_1#ps
-38	DecoupledStage$out_bits_2#ps
-38	DecoupledStage$out_bits_3#ps
-38	DecoupledStage$out_bits_4#ps
-38	DecoupledStage$out_bits_5#ps

End Points

Slack	Sink
-38	DecoupledPipe$DecoupledStage$out_bits_0#ns
-38	DecoupledPipe$DecoupledStage$out_bits_1#ns
-38	DecoupledPipe$DecoupledStage$out_bits_2#ns
-38	DecoupledPipe$DecoupledStage$out_bits_3#ns
-38	DecoupledPipe$DecoupledStage$out_bits_4#ns
-38	DecoupledPipe$DecoupledStage$out_bits_5#ns

Chisel Source

```
  va.io.in1 <> TypeRepacker(DecoupledStage(accin1.io.acc_out), to = va.io.in1.bits)
  va.io.in2 <> TypeRepacker(DecoupledStage(accin2.io.acc_out), to = va.io.in2.bits)

  accout.io.acc_data_in <> TypeRepacker(DelayModel(DecoupledStage(va.io.out),10, 2), to = accout.io.acc_data_in.bits)
  //accout.io.acc_data_in <> TypeRepacker(DecoupledStage(va.io.out), to = accout.io.acc_data_in.bits)

  accin1.io.acc_in <> io.acc_rd_req1
  accin2.io.acc_in <> io.acc_rd_req2
  accout.io.acc_req_in <> io.acc_wr_req

}

class VecAddMulDP[T <: UInt](gen : T, vecLen : Int)(implicit params : AccParams) extends SDFActor(2,1) {
    override lazy val io = IO(new ActorIO {
      val in1 = In(Vec(vecLen, gen), 1)
      val in2 = In(Vec(vecLen, gen), 1)
      val out = Out(Vec(vecLen, gen), 1)
    })

    override def func = {
      val add = Vec.tabulate(vecLen) {i => io.in1.bits(i) + io.in2.bits(i) }
      val mul1 = Vec.tabulate(vecLen) {i => io.in1.bits(i) * add(i) }
      val mul2 = Vec.tabulate(vecLen) {i => io.in2.bits(i) * add(i) }
      val add2 = Vec.tabulate(vecLen) {i => mul1(i) + mul2(i) }
      val mul3 = Vec.tabulate(vecLen) {i => mul1(i) * add2(i) }
      val mul4 = Vec.tabulate(vecLen) {i => mul2(i) * add2(i) }
      val add3 = Vec.tabulate(vecLen) {i => mul3(i) + mul4(i) }
      io.out.bits := add3
    }
}

class VecAddMulDPWStages[T <: UInt](gen : T, vecLen : Int)(implicit params : AccParams) extends Module {

  val io = IO(new ActorIO {
    val in0 = In(UInt((gen.getWidth*vecLen).W), 1)
```

Op	Incr	Required	Net
connect	0	38	DecoupledStage$out_bits_0#ps
connect	0	38	DecoupledStage$io_out_bits_0
add,((16,16))	5	38	va$io_in1_bits_0
tail,((17))	0	33	va$_T_254
connect	0	33	va$_T_255
mul,((16,16))	8	33	va$_T_272_0
connect	0	25	va$_T_304
add,((32,32))	6	25	va$_T_314_0
tail,((33))	0	19	va$_T_325
connect	0	19	va$_T_326
mul,((32,32))	10	19	va$_T_343_0
connect	0	9	va$_T_354
add,((64,64))	7	9	va$_T_364_0
tail,((65))	0	2	va$_T_396
connect	0	2	va$_T_397
mux,((1),(64))	1	2	va$_T_414_0

What had to change?

A Chisel Compiler

CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

Stable and User-Friendly API

External Collaborations

Emphasis on Clarity

Chisel 2.0.0

```
Reg(UInt(3))
```

1. Register of type UInt, width of 3
2. Register whose next cycle's value is 3
3. Register whose initial value is 3
4. Register no initial value and width of 2

Chisel 3.0.0

```
Reg(UInt(3.W))
RegNext(3.U)
RegInit(3.U)
Reg(chiselTypeOf(3.U))
```

1. Register of type UInt, width of 3
2. Register whose next cycle's value is 3
3. Register whose initial value is 3
4. Register no initial value and width of 2

Less Error-Prone API’s

Chisel 2.0.0

```
def func[T<:Chisel.Data](other: T) = {
  io.out := Reg(null.asInstanceOf[T], next = other, null.asInstanceOf[T])
}
```

```
[info] [0.000] Elaborating design...
[info] [0.022] Done elaborating.
```

Chisel 3.0.0

```
def func[T<:Chisel.Data](other: T) = {
  io.out := RegNext(other)
}
```

```
[info] [0.000] Elaborating design...
[info] [0.022] Done elaborating.
```

Lightweight Support for Multi-Clock/Reset

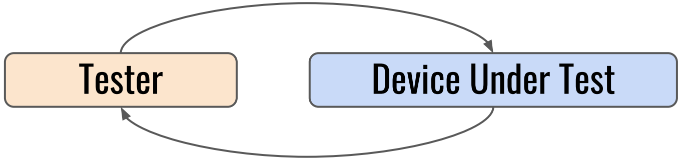
```
withReset(io.alternateReset) {
      val altRst = RegInit(0.U(10.W))
      …
}
```

Everything in this scope with have io.alternateReset as the reset

```
withClock(io.alternateClock) {
      val altClk = RegInit(0.U(10.W))
      …
}
```

Everything in this scope with have io.alternateClock as the clock

Exciting Future Work: The Unified Chisel Tester



The following image is a DRAMATIZATION of real experiences

(Current Chisel Testing Environment)

- Fragmented Landscape
  - BasicTester (Hardware Testing Hardware)
  - PeekPokeTester (Interactive and Slow)
  - AdvancedTester (Limited Concurrency)
- Unified Chisel Tester
  - Lightweight, powerful, fast
  - Multiple circuit drivers, multithreaded
  - Integration with Verilator, VCS, Interpreter
- If you have thoughts - send them our way!

What had to change?

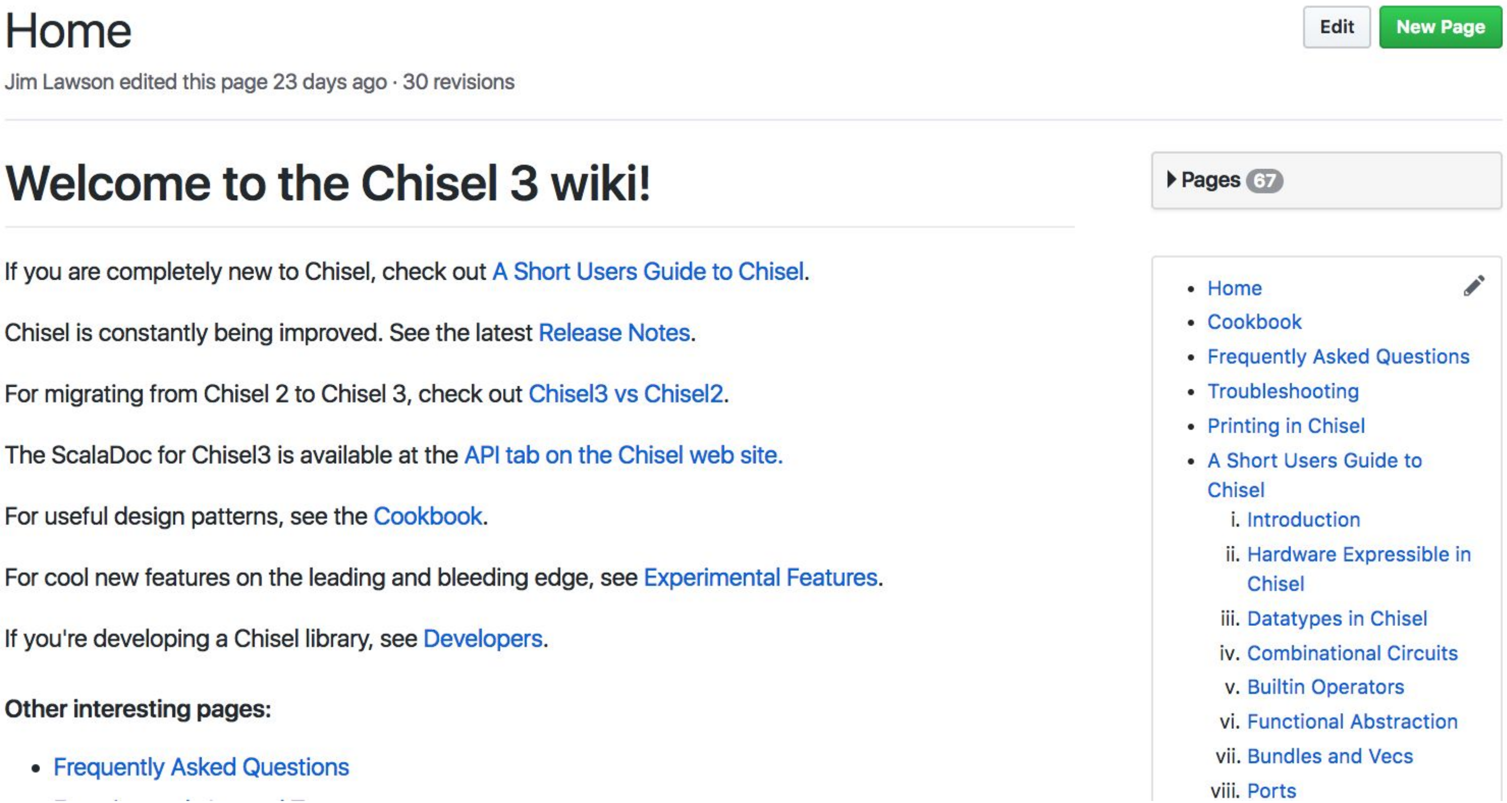
A Chisel Compiler

CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

Stable and User-Friendly API

External Collaborations

Documentation, Documentation, Documentation



Home

Jim Lawson edited this page 23 days ago · 30 revisions

Welcome to the Chisel 3 wiki!

If you are completely new to Chisel, check out A Short Users Guide to Chisel.

Chisel is constantly being improved. See the latest Release Notes.

For migrating from Chisel 2 to Chisel 3, check out Chisel3 vs Chisel2.

The ScalaDoc for Chisel3 is available at the API tab on the Chisel web site.

For useful design patterns, see the Cookbook.

For cool new features on the leading and bleeding edge, see Experimental Features.

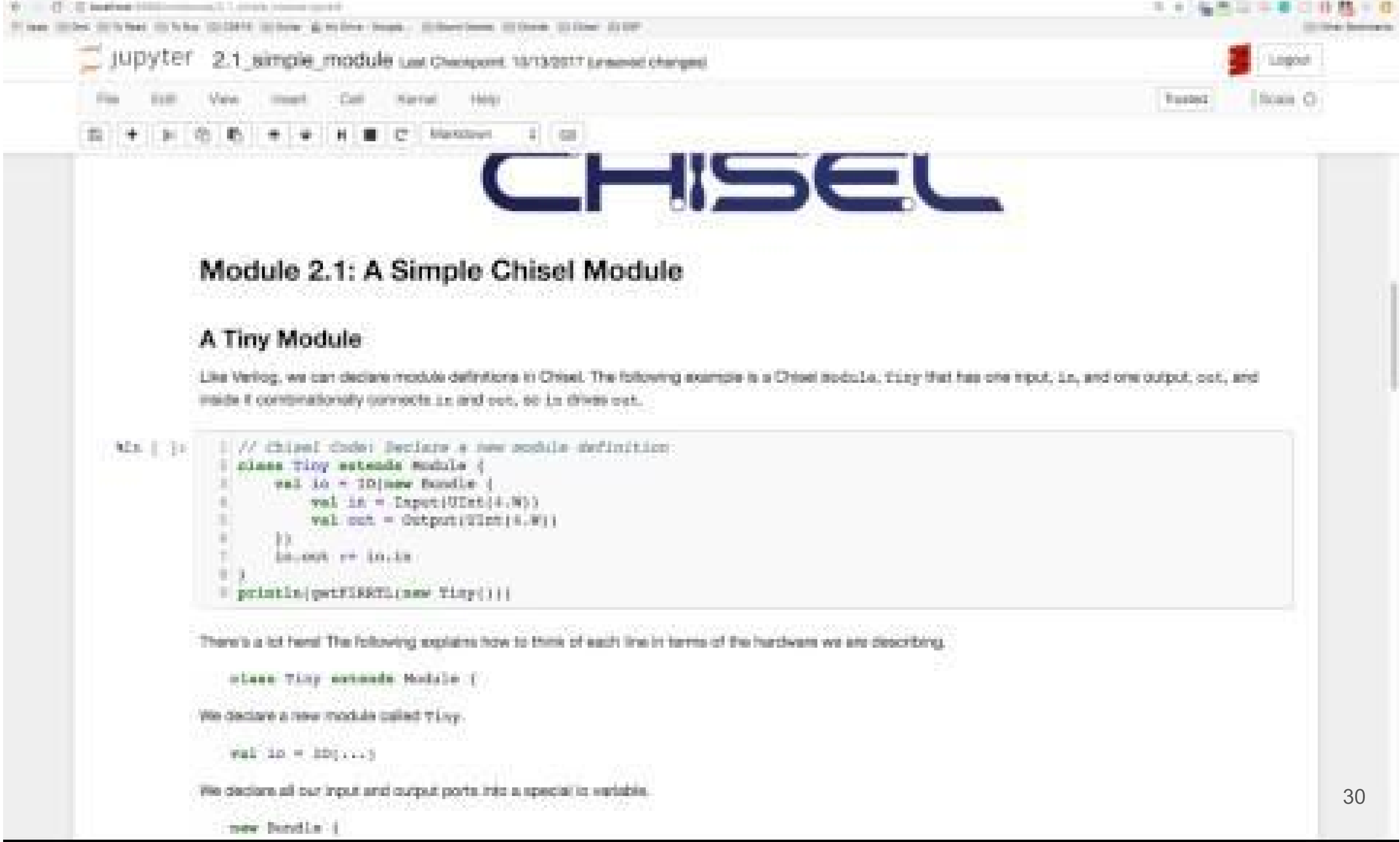
If you're developing a Chisel library, see Developers.

Other interesting pages:

- Frequently Asked Questions

- Home
- Cookbook
- Frequently Asked Questions
- Troubleshooting
- Printing in Chisel
- A Short Users Guide to Chisel
  - i. Introduction
  - ii. Hardware Expressible in Chisel
  - iii. Datatypes in Chisel
  - iv. Combinational Circuits
  - v. Builtin Operators
  - vi. Functional Abstraction
  - vii. Bundles and Vecs
  - viii. Ports

Bootcamps and Tutorials



Module 2.1: A Simple Chisel Module

A Tiny Module

Like Verilog, we can declare module definitions in Chisel. The following example is a Chisel `Module`, `Tiny` that has one input, `in`, and one output, `out`, and inside it combinationally connects `in` and `out`, so `in` drives `out`.

In [ ]:

```
// Chisel Code: Declare a new module definition
class Tiny extends Module {
    val io = IO(new Bundle {
        val in = Input(UInt(4.W))
        val out = Output(UInt(4.W))
    })
    io.out := io.in
}
println(getFIRRTL(new Tiny()))
```

There's a lot here! The following explains how to think of each line in terms of the hardware we are describing.

```
class Tiny extends Module {
```

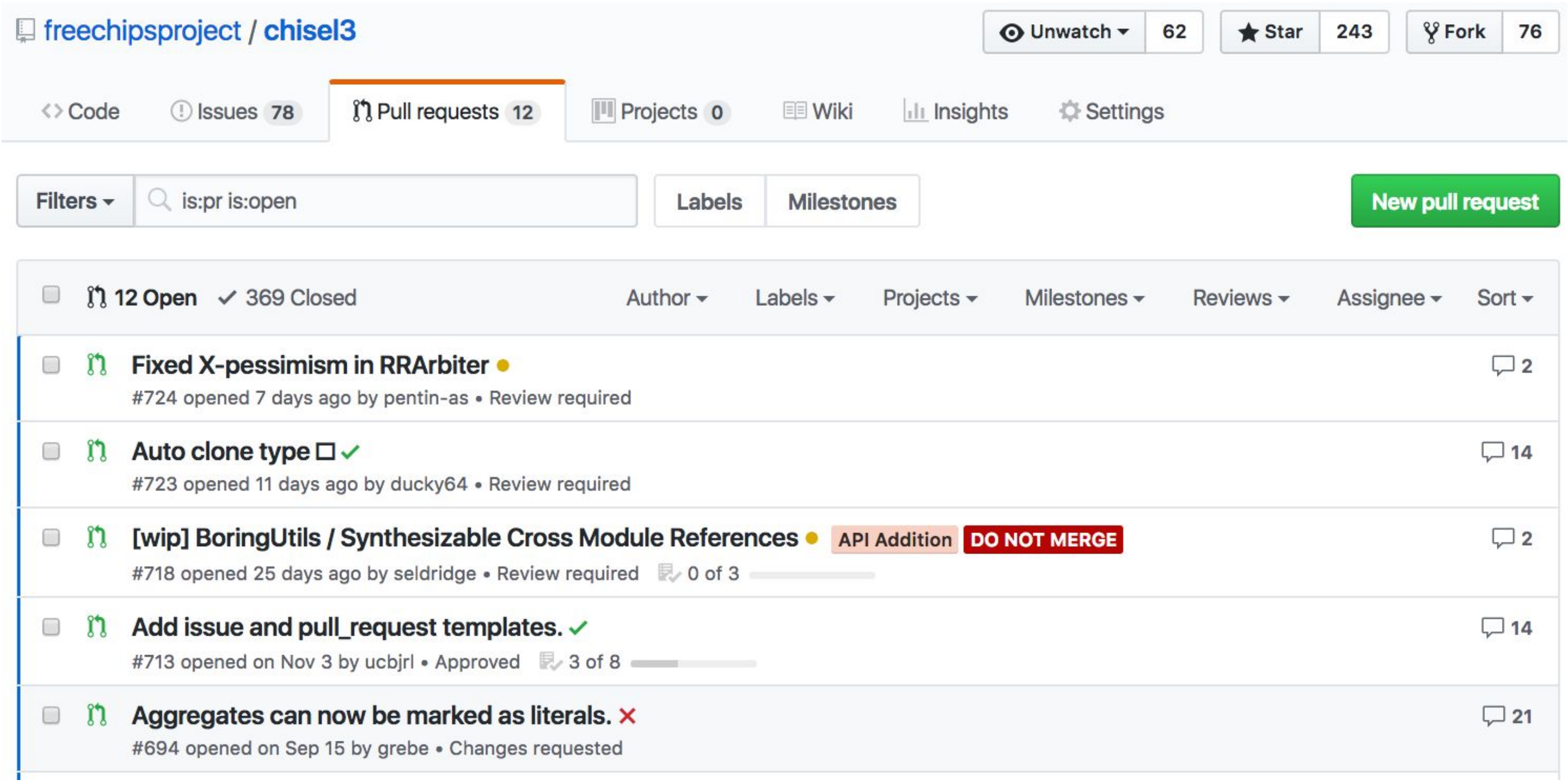
We declare a new module called `Tiny`.

```
val io = IO(...)
```

We declare all our input and output ports into a special `io` variable.

```
new Bundle {
```

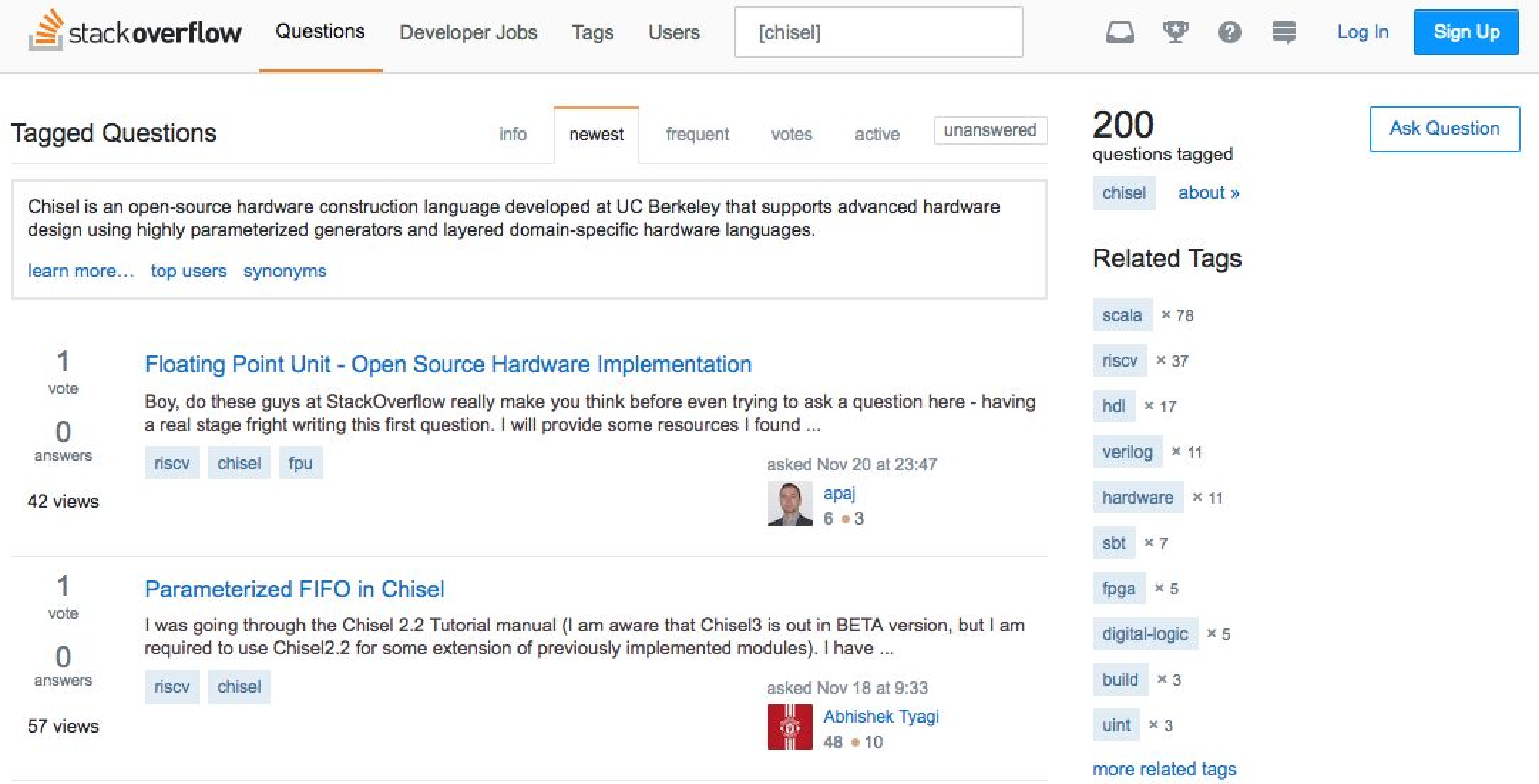
Open-Development via Github Issues/Pull Requests



is:pr is:open

Fixed X-pessimism in RRArbiter ● #724 opened 7 days ago by pentin-as • Review required	2
Auto clone type □ ✓ #723 opened 11 days ago by ducky64 • Review required	14
[wip] BoringUtils / Synthesizable Cross Module References ● API Addition DO NOT MERGE #718 opened 25 days ago by seldridge • Review required 0 of 3	2
Add issue and pull_request templates. ✓ #713 opened on Nov 3 by ucbjrl • Approved 3 of 8	14
Aggregates can now be marked as literals. ✗ #694 opened on Sep 15 by grebe • Changes requested	21

Stack Overflow!



Tagged Questions

Chisel is an open-source hardware construction language developed at UC Berkeley that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages.

learn more… top users synonyms

1 vote 0 answers 42 views

Floating Point Unit - Open Source Hardware Implementation

Boy, do these guys at StackOverflow really make you think before even trying to ask a question here - having a real stage fright writing this first question. I will provide some resources I found ...

riscv chisel fpu

asked Nov 20 at 23:47 apaj 6 ● 3

1 vote 0 answers 57 views

Parameterized FIFO in Chisel

I was going through the Chisel 2.2 Tutorial manual (I am aware that Chisel3 is out in BETA version, but I am required to use Chisel2.2 for some extension of previously implemented modules). I have ...

riscv chisel

asked Nov 18 at 9:33 Abhishek Tyagi 48 ● 10

200 questions tagged

chisel about »

Related Tags

Tag	Count
scala	× 78
riscv	× 37
hdl	× 17
verilog	× 11
hardware	× 11
sbt	× 7
fpga	× 5
digital-logic	× 5
build	× 3
uint	× 3

more related tags

Academic Impact: 188 citations (in 5 years)

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.
Copyright 2012 ACM 978-1-4503-1199-1/12/06 ...$10.00.

Berkeley
UNIVERSITY OF CALIFORNIA

IBM®

Stanford
University

Microsoft®
Research

Cornell University

DTU
Technical University of Denmark

UNIVERSITY OF CAMBRIDGE

TEXAS TECH
UNIVERSITY®

Massachusetts Institute of Technology

CARNEGIE MELLON UNIVERSITY
PITTSBURGH PENNSYLVANIA 1900

NTNU
Norwegian University of Science and Technology

Nara Institute of Science and Technology
NAIST®

BERKELEY LAB
Lawrence Berkeley National Laboratory

W
UNIVERSITY *of* WASHINGTON

BOSTON UNIVERSITY

Active Users (That We Know Of)

SiFive

Stanford University

DRAPER

Berkeley Architecture Research

Google

intel®

Massachusetts Institute of Technology

IBM®

DTU

Technical University of Denmark

BERKELEY LAB

Lawrence Berkeley National Laboratory

BAE SYSTEMS

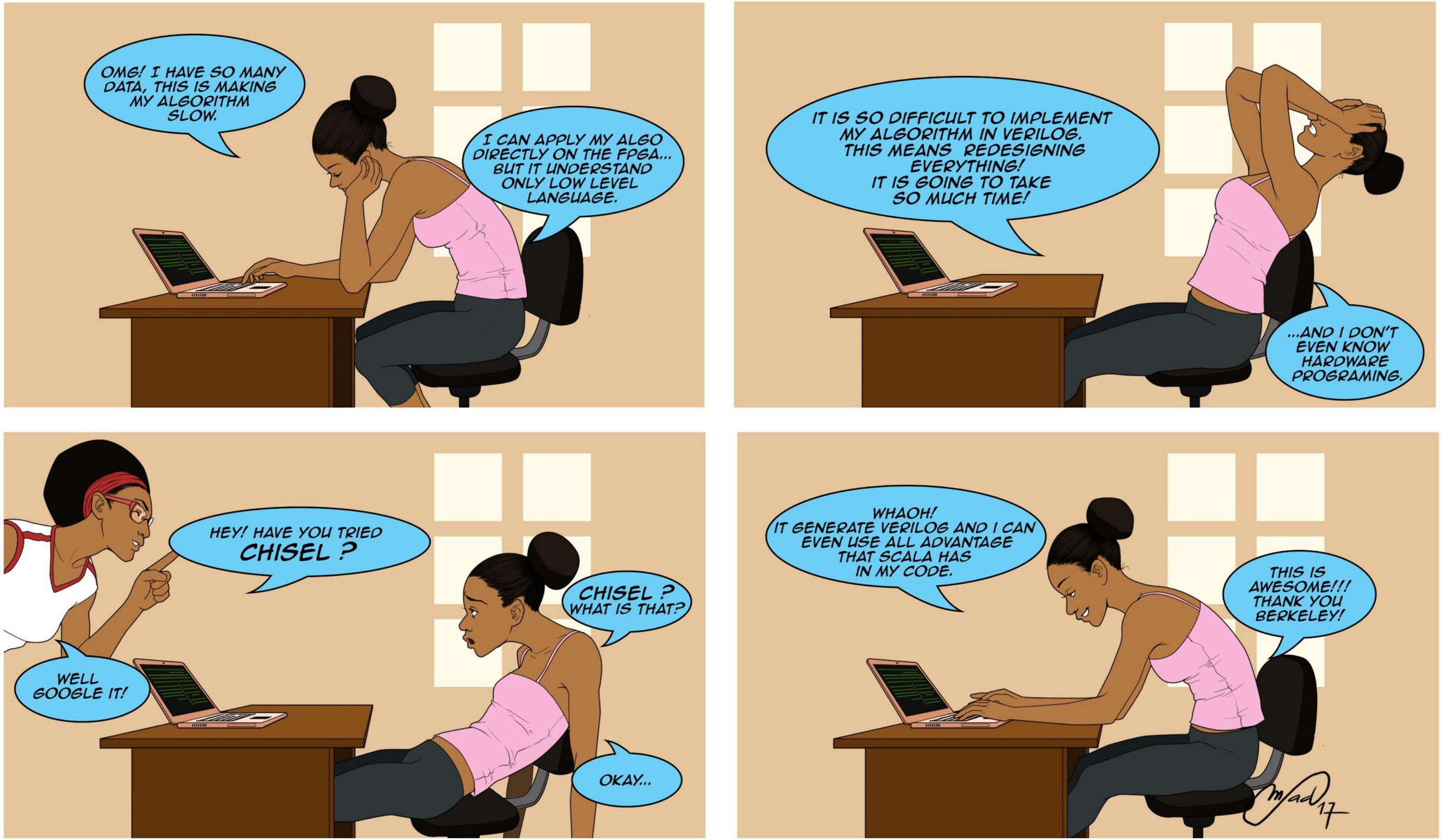
INSPIRED WORK

Yale

lowRISC

OPEN TO THE CORE

BWRC


OMG! I HAVE SO MANY DATA, THIS IS MAKING MY ALGORITHM SLOW.
I CAN APPLY MY ALGO DIRECTLY ON THE FPGA... BUT IT UNDERSTAND ONLY LOW LEVEL LANGUAGE.
IT IS SO DIFFICULT TO IMPLEMENT MY ALGORITHM IN VERILOG. THIS MEANS REDESIGNING EVERYTHING! IT IS GOING TO TAKE SO MUCH TIME!
...AND I DON'T EVEN KNOW HARDWARE PROGRAMING.
HEY! HAVE YOU TRIED CHISEL ?
CHISEL ? WHAT IS THAT?
WELL GOOGLE IT!
OKAY...
WHAOH! IT GENERATE VERILOG AND I CAN EVEN USE ALL ADVANTAGE THAT SCALA HAS IN MY CODE.
THIS IS AWESOME!!! THANK YOU BERKELEY!

Ideas for Governance? Maintenance? Workshops?

Thanks to all Chiselers out there! (And many more!!)

So long (ASPIRE), and thanks for all the ~~fish~~ chips!