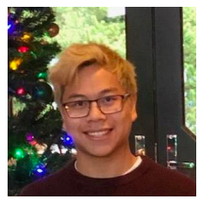
Generate Compilers from Hardware Models!

Gus Smith, PhD Candidate, University of Washington
PLARCH 2023

Hi everybody. My name is Gus Smith, and I'm a PhD candidate at the University of Washington's PLSE lab. Today I'm going to be talking about why we should generate compilers from hardware models.


Ben Kushigian

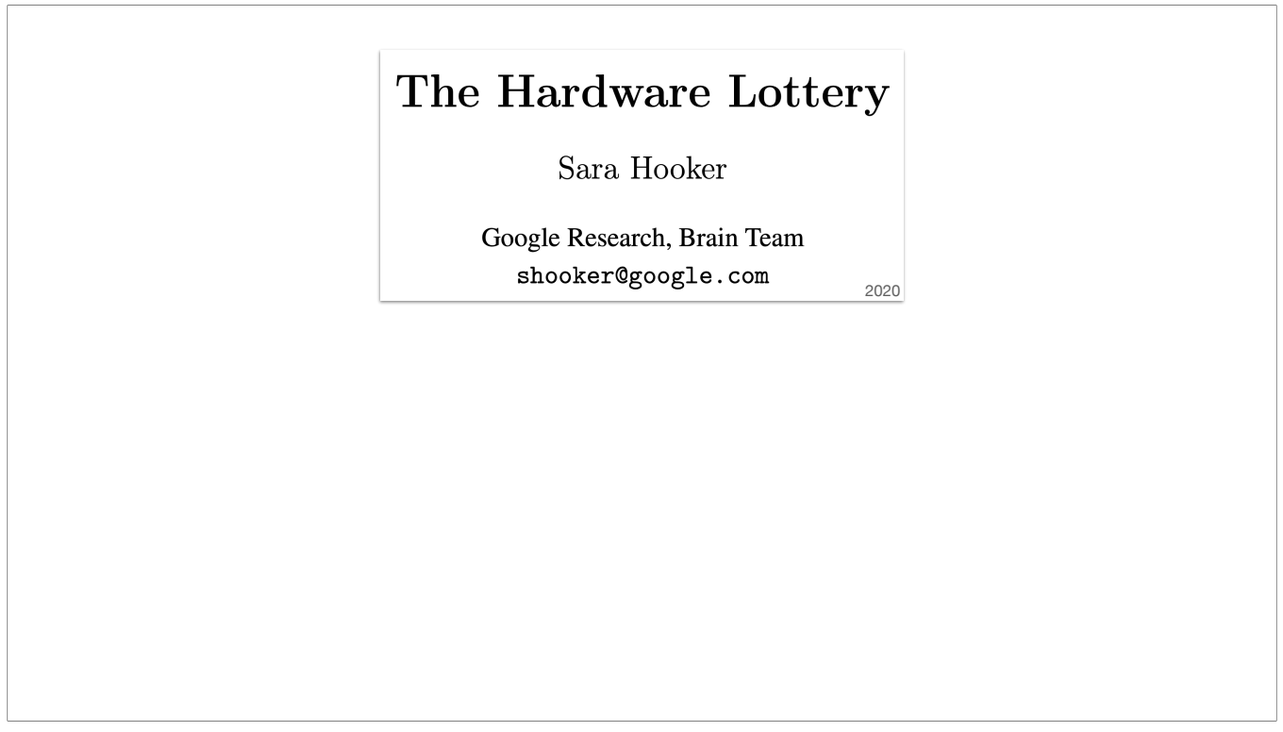

Vishal Canumalla


Andrew Cheung


René Just


Zachary Tatlock

First, let me shout out my coauthors in PLSE: Ben, Vishal, Andrew, Rene, and Zach, some of whom are in the audience today!



In 2020, Sara Hooker from Google Research released a paper titled the Hardware Lottery. The thesis of this paper is that

[build] …

She argues that, for example, the current track of matrix-multiplication-based advances in machine learning are undeniably linked to the abundance of hardware for matrix multiplication, and that other research directions in machine learning are subsequently less likely to be successful.

The hardware lottery is a direct challenge to us in this room. By my reading, the takeaway for our community is that
[build] new platforms (that is, new hardware and their associated compiler stack) should be easier to build, so that the best ideas win — not just the ideas with hardware on their side!

In this talk, I'm going to focus on what I know:
[build] compilers.
So, let's ask the question,
[build] why *are* compilers so difficult

The Hardware Lottery

Sara Hooker

Google Research, Brain Team
shooker@google.com

2020

Hardware and compilers have a disproportionate role in deciding which research ideas succeed or fail.

The Hardware Lottery

Sara Hooker

Google Research, Brain Team
shooker@google.com

2020

Hardware and compilers have a disproportionate role in deciding which research ideas succeed or fail.

Takeaway for our community: new platforms (hardware + compiler stack) should be easier to build, so that the best ideas win!

The Hardware Lottery

Sara Hooker

Google Research, Brain Team
shooker@google.com

2020

Hardware and compilers have a disproportionate role in deciding which research ideas succeed or fail.

Takeaway for our community: new platforms (hardware + compiler stack) should be easier to build, so that the best ideas win!

The Hardware Lottery

Sara Hooker

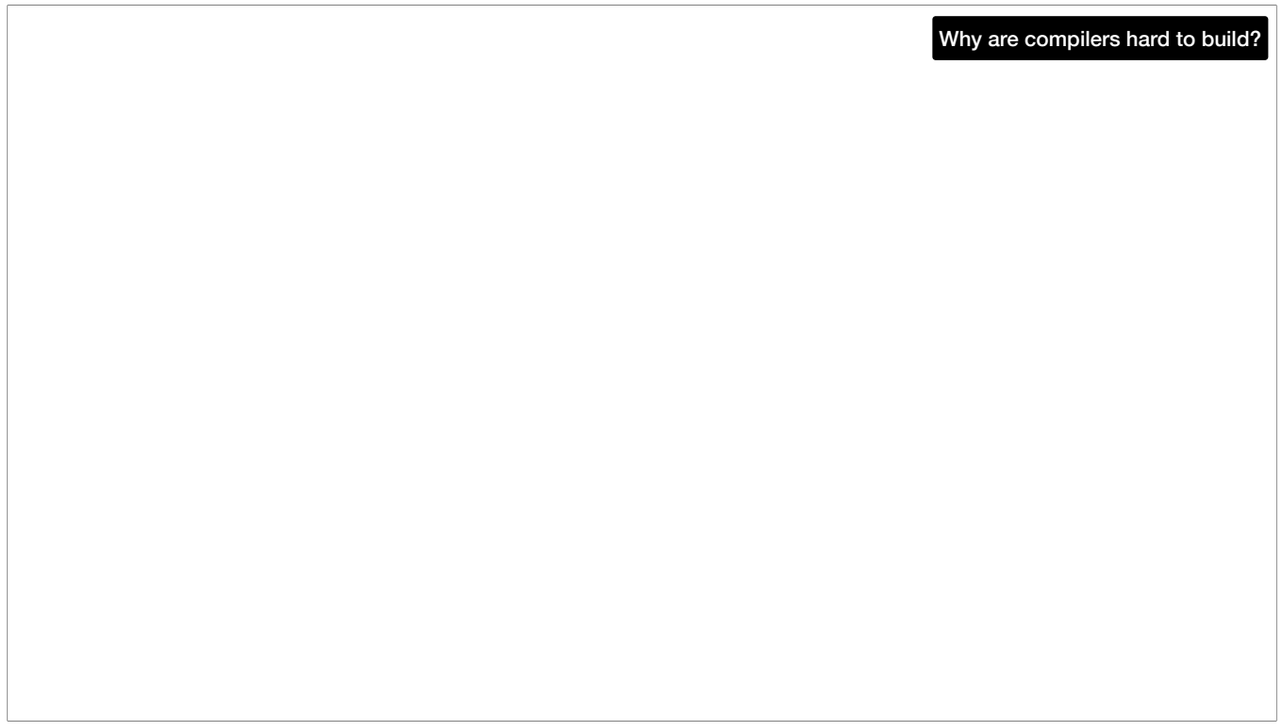
Google Research, Brain Team
shooker@google.com

2020

Hardware and compilers have a disproportionate role in deciding which research ideas succeed or fail.

Takeaway for our community: new platforms (hardware + compiler stack) should be easier to build, so that the best ideas win!

Why are compilers hard to build?

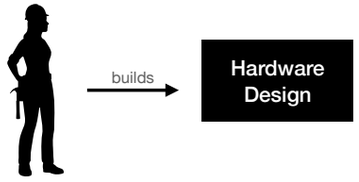


Imagine we have
[build] a hardware engineer who
[build] builds a new hardware design. If she wants to run
[build] programs on her hardware, she'll need a
[build] compiler that
[build] compiles programs to run on her design. This requires
[build] a compiler engineer, who, with much time and effort
[build] builds a compiler, whose design is informed not only by
[build] the hardware design itself, but also by
[build] communications with the hardware designer,
[build] any documentation that exists for the design, and finally, by
[build] the compiler engineer's own internal model of how the hardware works. If that's not confusing enough, if the team wants to ensure the compiler is correct, they'll hire
[build] a verification engineer to
[build] build a
[build] formal verification model, which
[build] models the hardware and
[build] verifies that the compiler is correct. Similarly to the compiler, the design of the verification model is informed by
[build] communications with the hardware designer,
[build] the hardware design itself,

[build] the documentation, and
[build] the verification engineer’s own ideas about how the hardware works.
[build] If this is looking a little complicated, well, I agree!
But the process of building a compiler is more than just confusing; it also
[build] requires significant developer effort,

Why are compilers hard to build?

Why are compilers hard to build?



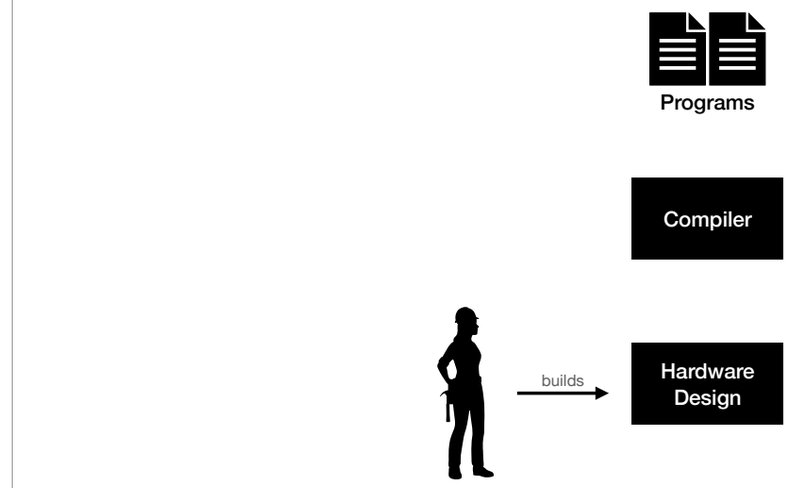
Why are compilers hard to build?

Programs

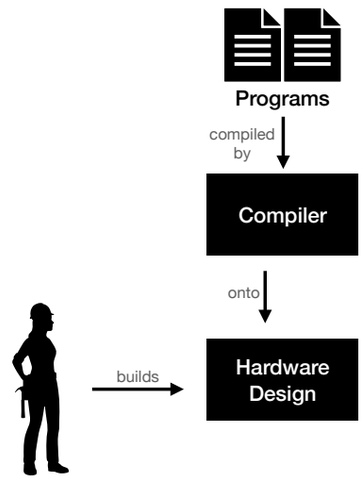
builds

Hardware Design

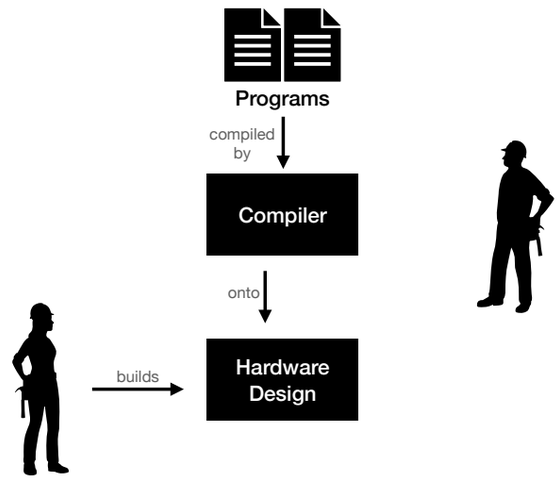
Why are compilers hard to build?



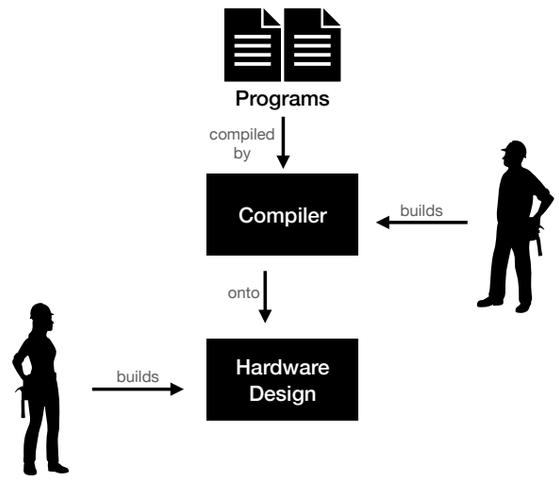
Why are compilers hard to build?



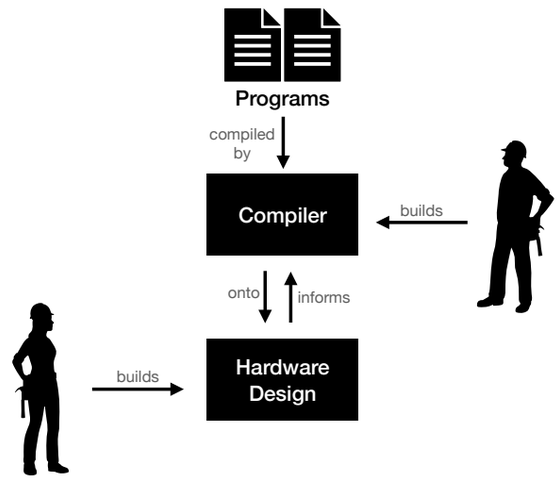
Why are compilers hard to build?



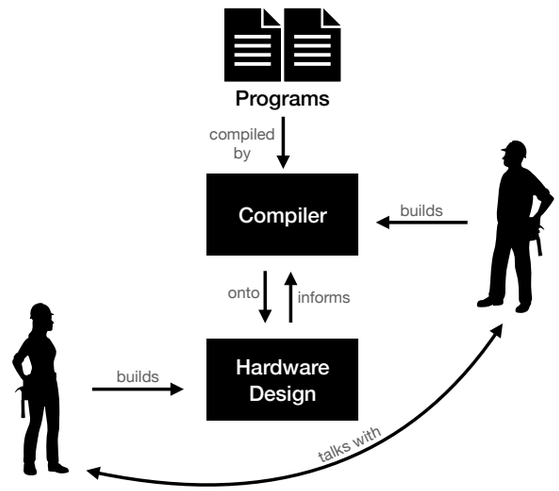
Why are compilers hard to build?

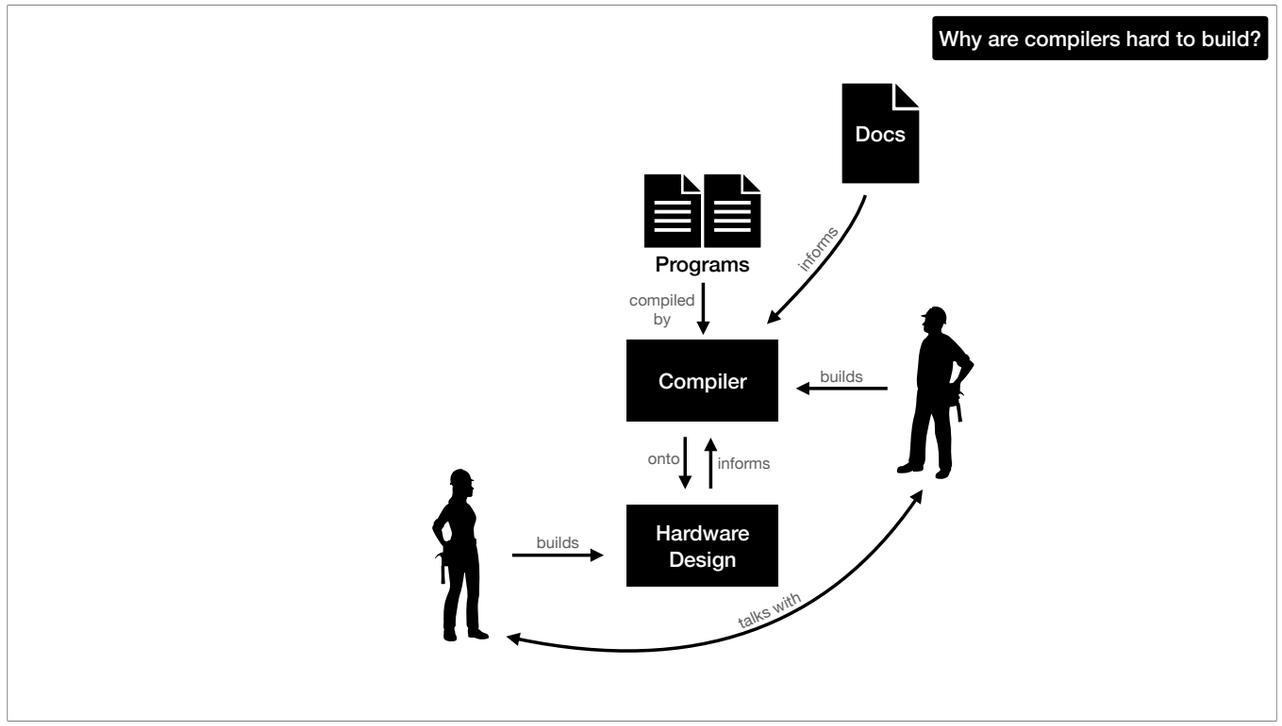


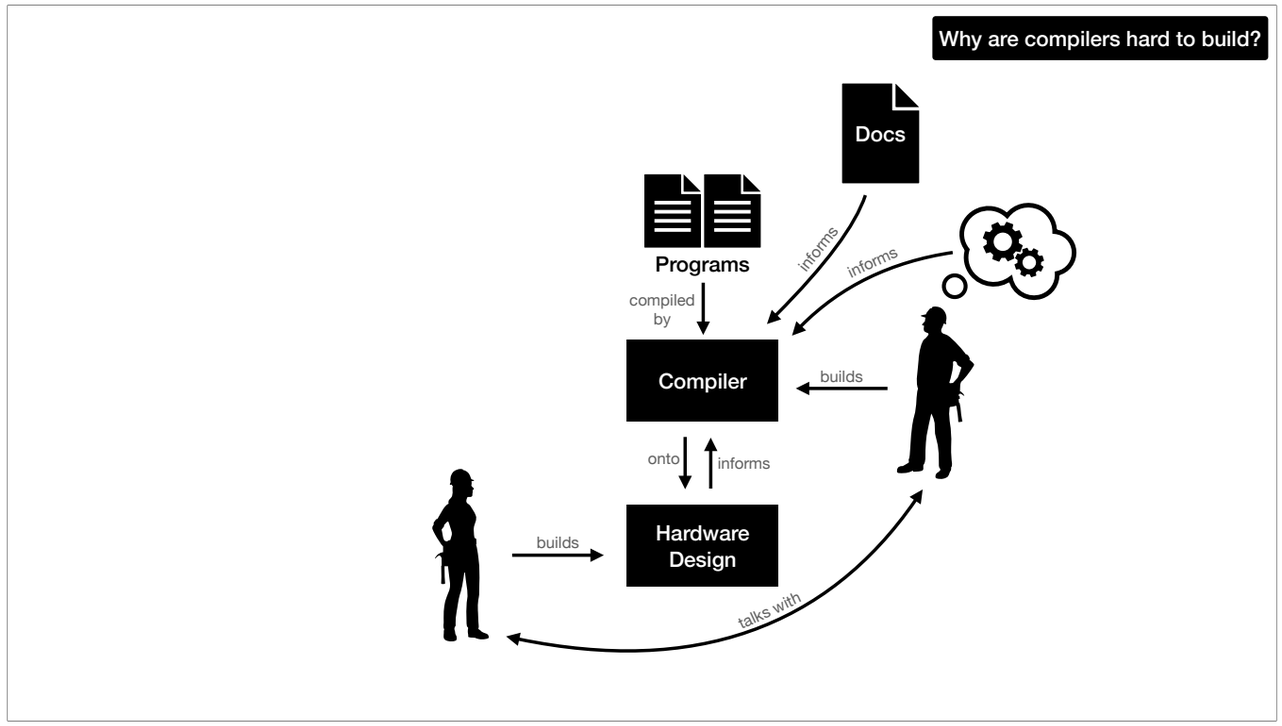
Why are compilers hard to build?

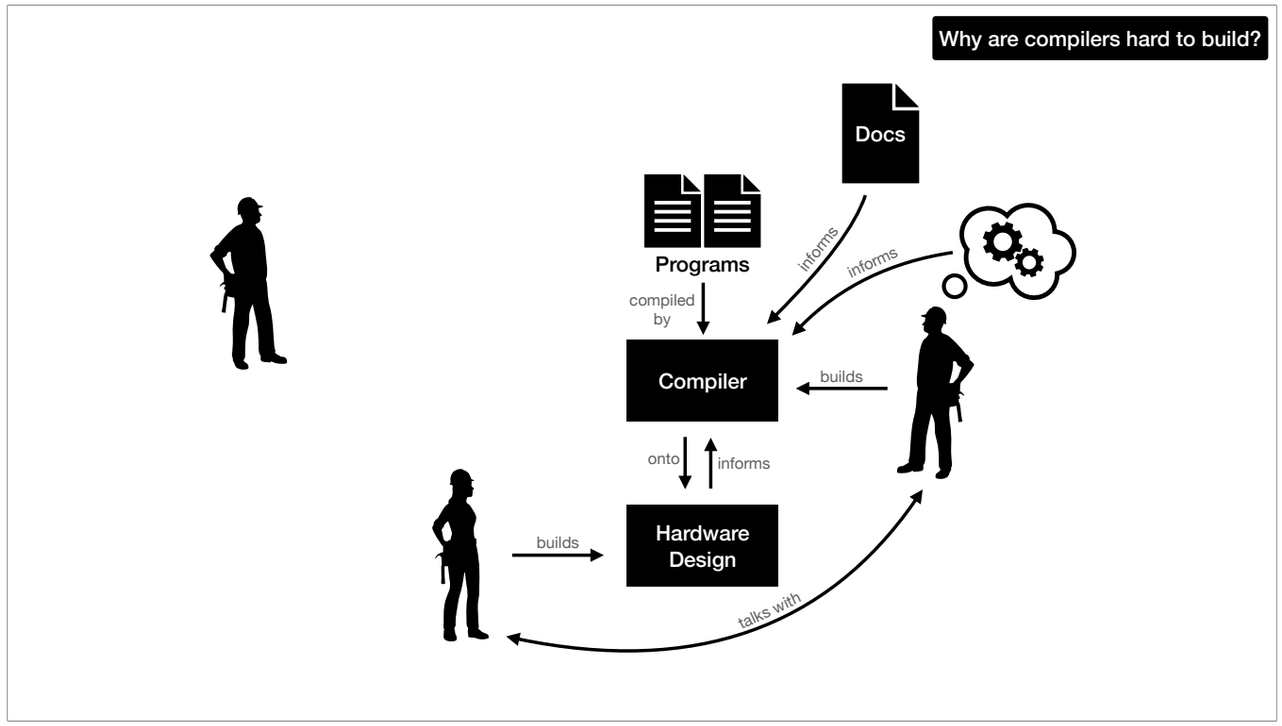


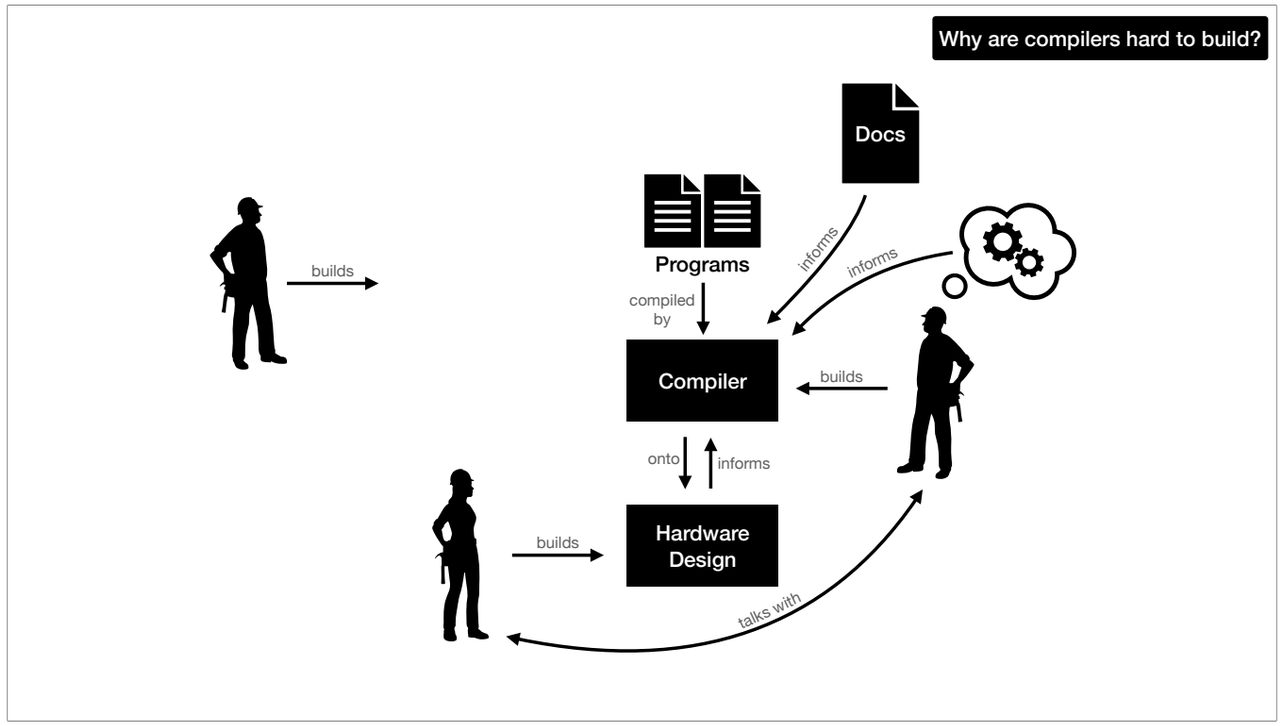
Why are compilers hard to build?

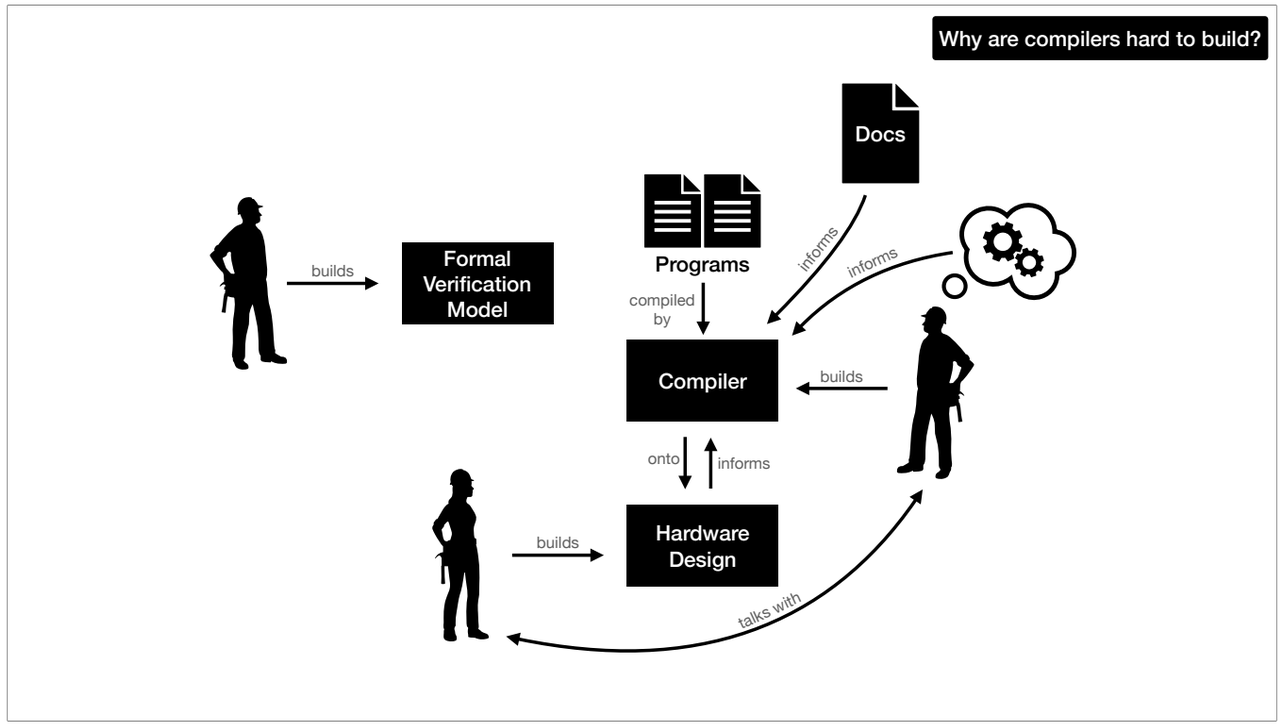


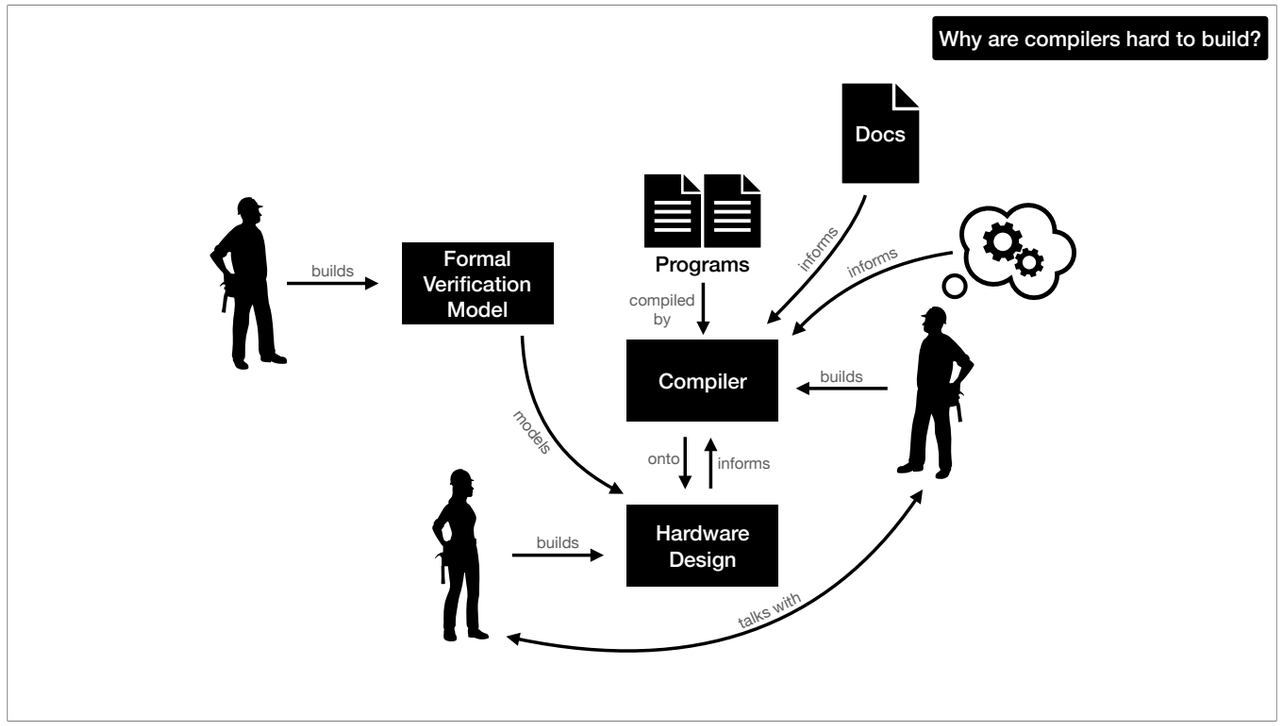

Why are compilers hard to build?
Docs
Programs
informs
compiled by
Compiler
builds
onto
informs
builds
Hardware Design
talks with

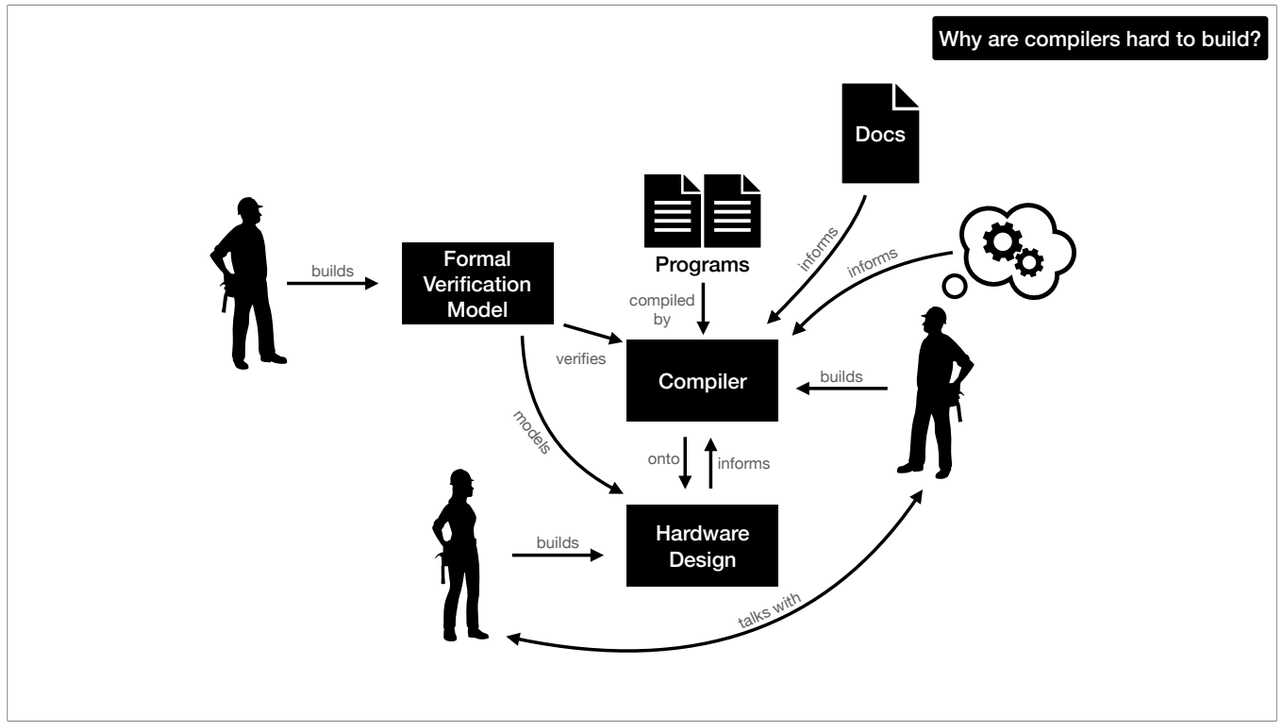

Why are compilers hard to build?
Docs
Programs
informs
informs
compiled by
Compiler
builds
onto
informs
builds
Hardware Design
talks with

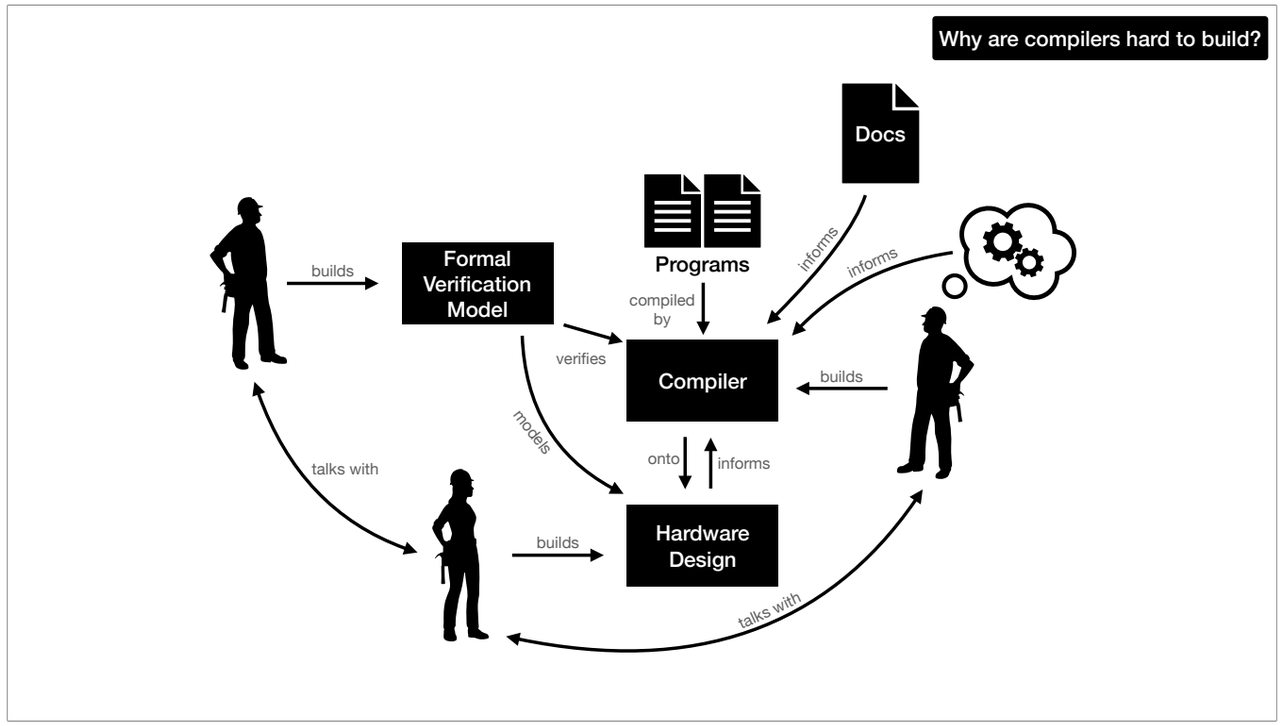

Why are compilers hard to build?
Docs
Programs
informs
informs
compiled by
Compiler
builds
onto
informs
builds
Hardware Design
talks with

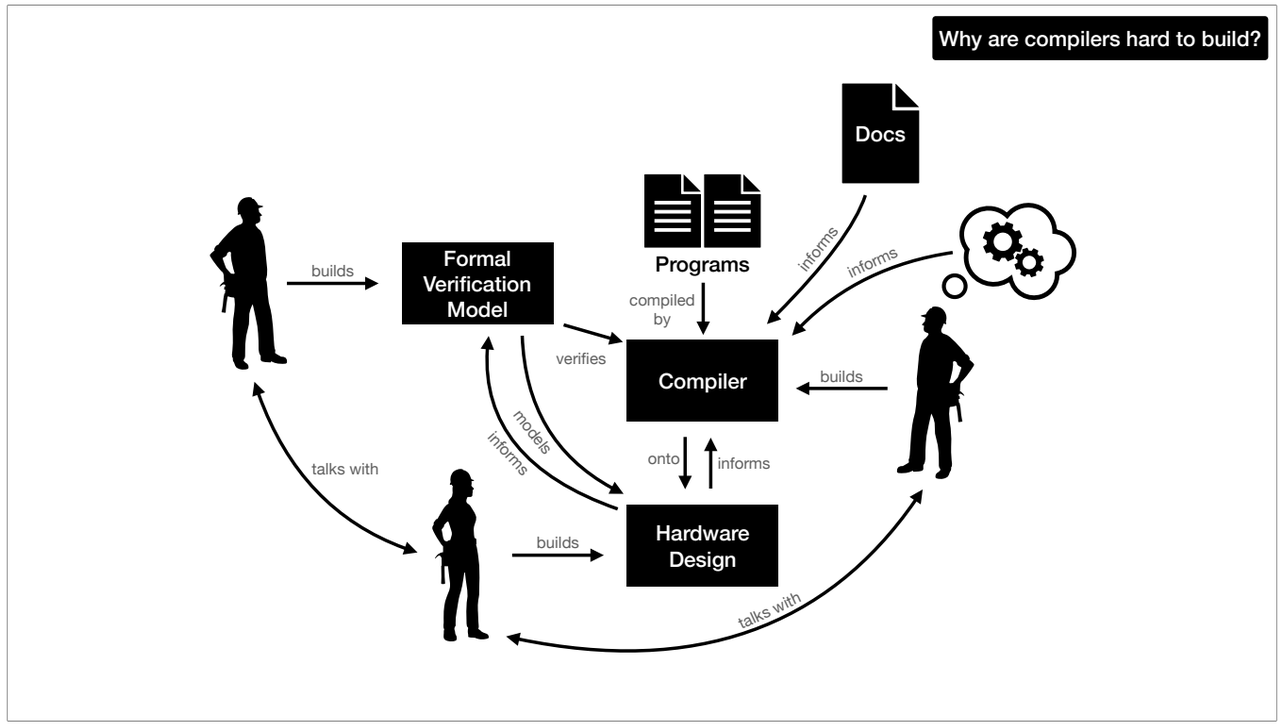

Why are compilers hard to build?
Docs
Programs
builds
informs
informs
compiled by
Compiler
builds
onto
informs
builds
Hardware Design
talks with

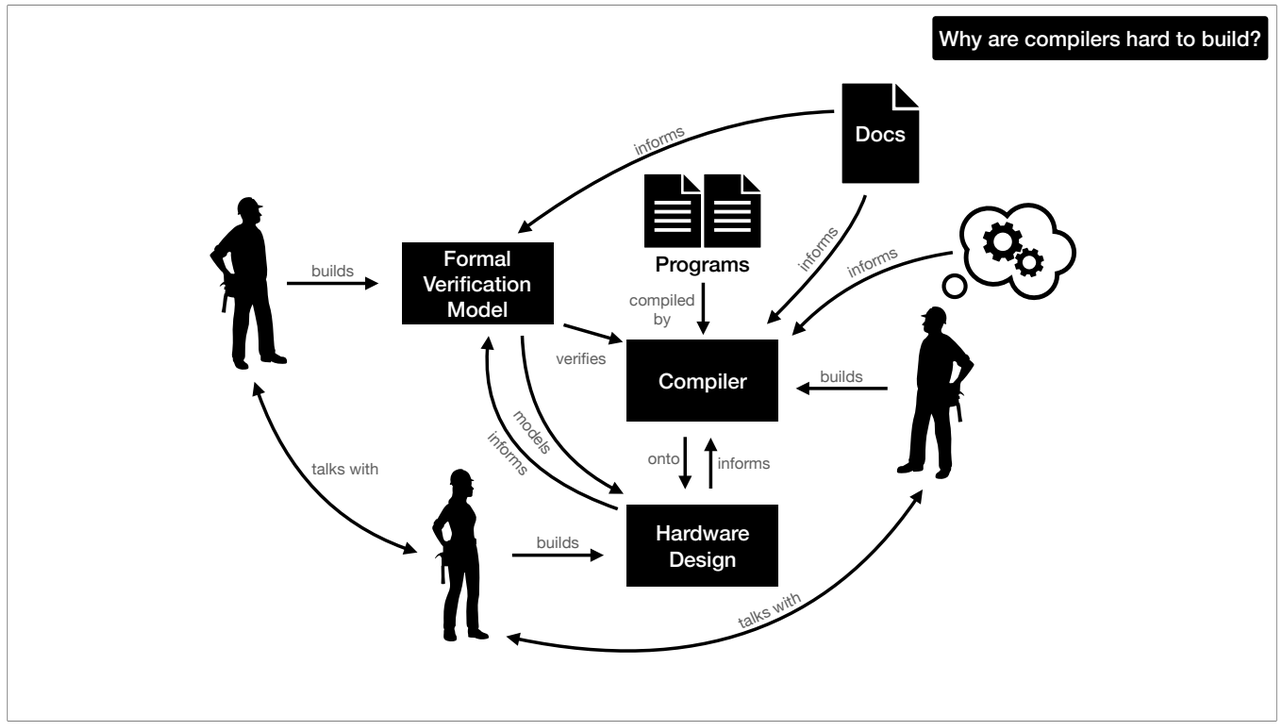

Why are compilers hard to build?
Docs
Programs
builds
Formal Verification Model
informs
informs
compiled by
Compiler
builds
onto
informs
builds
Hardware Design
talks with

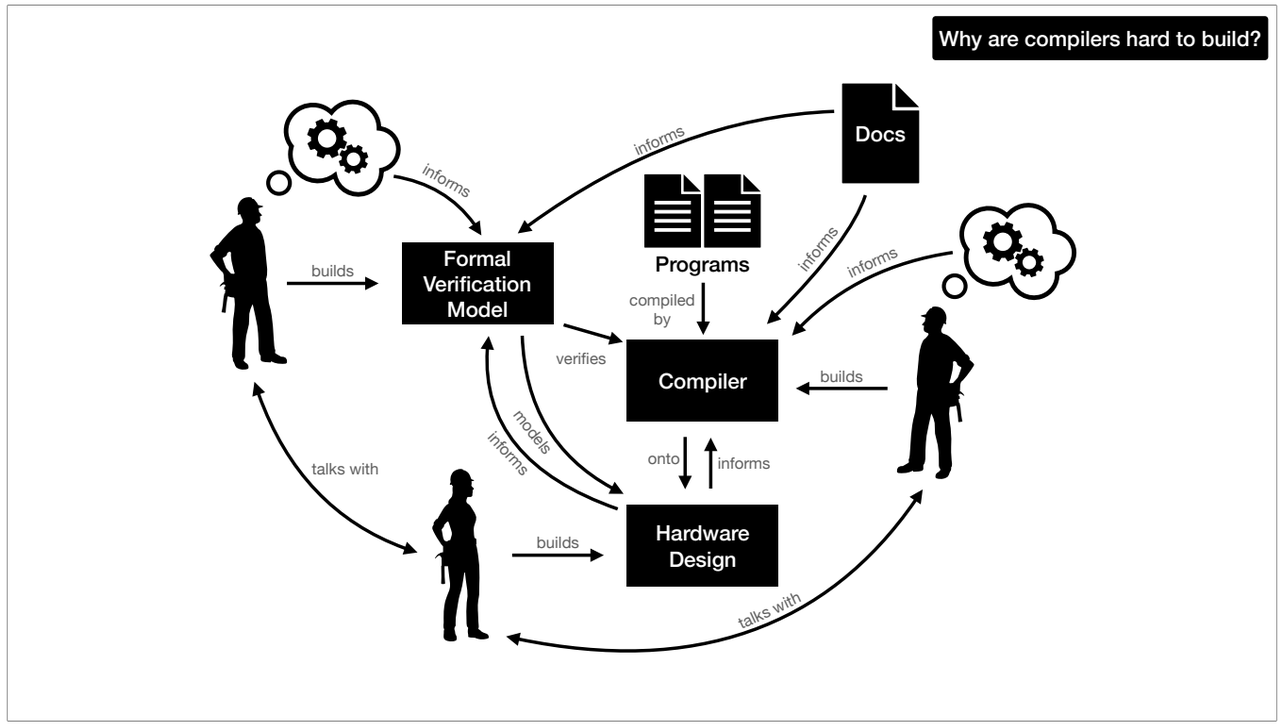

Why are compilers hard to build?
Docs
Programs
Formal Verification Model
builds
informs
informs
compiled by
Compiler
builds
models
onto
informs
builds
Hardware Design
talks with

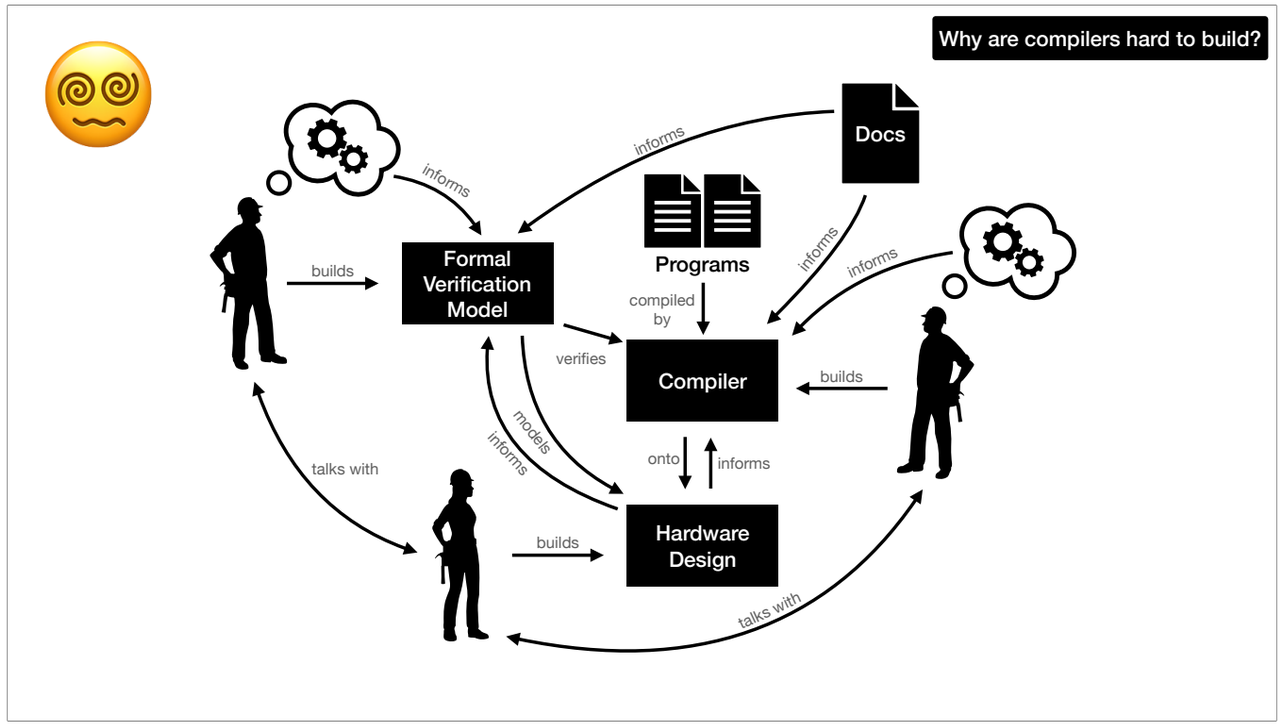

Why are compilers hard to build?
Docs
Programs
Formal Verification Model
builds
informs
informs
compiled by
verifies
Compiler
builds
models
onto
informs
Hardware Design
builds
talks with

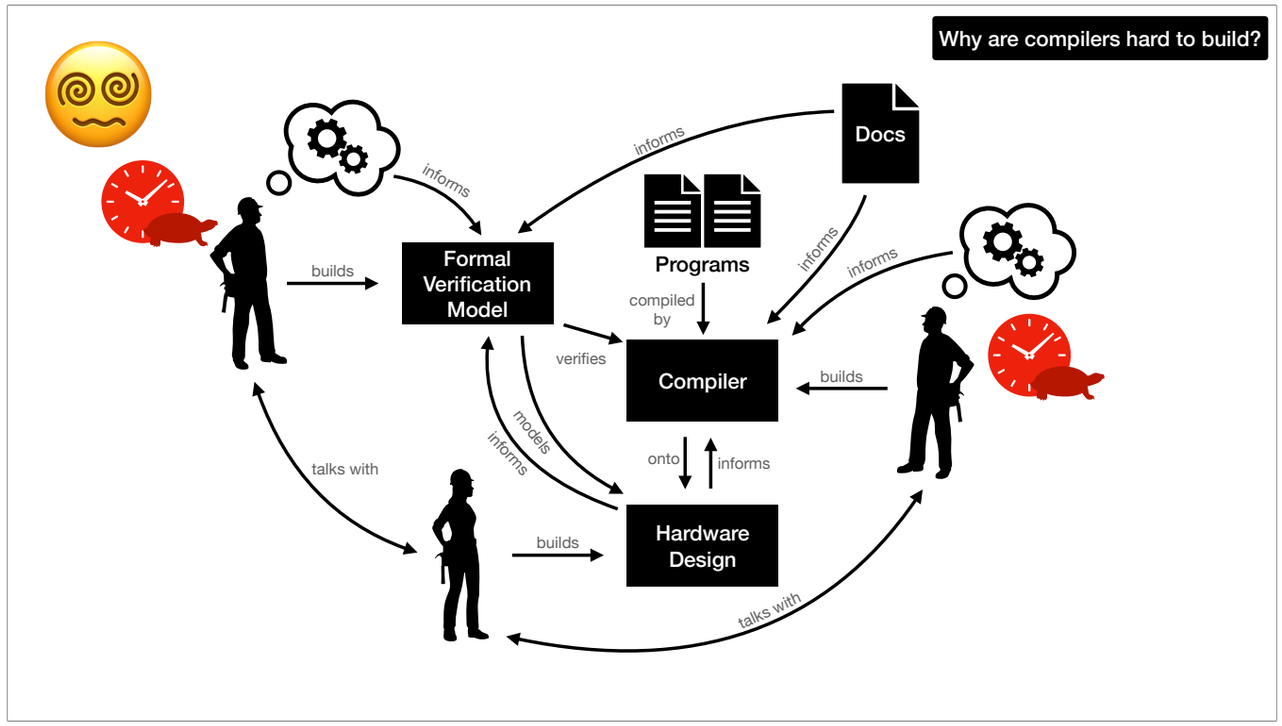

Why are compilers hard to build?
Docs
Programs
Formal Verification Model
builds
informs
informs
compiled by
verifies
Compiler
builds
models
onto
informs
talks with
builds
Hardware Design
talks with

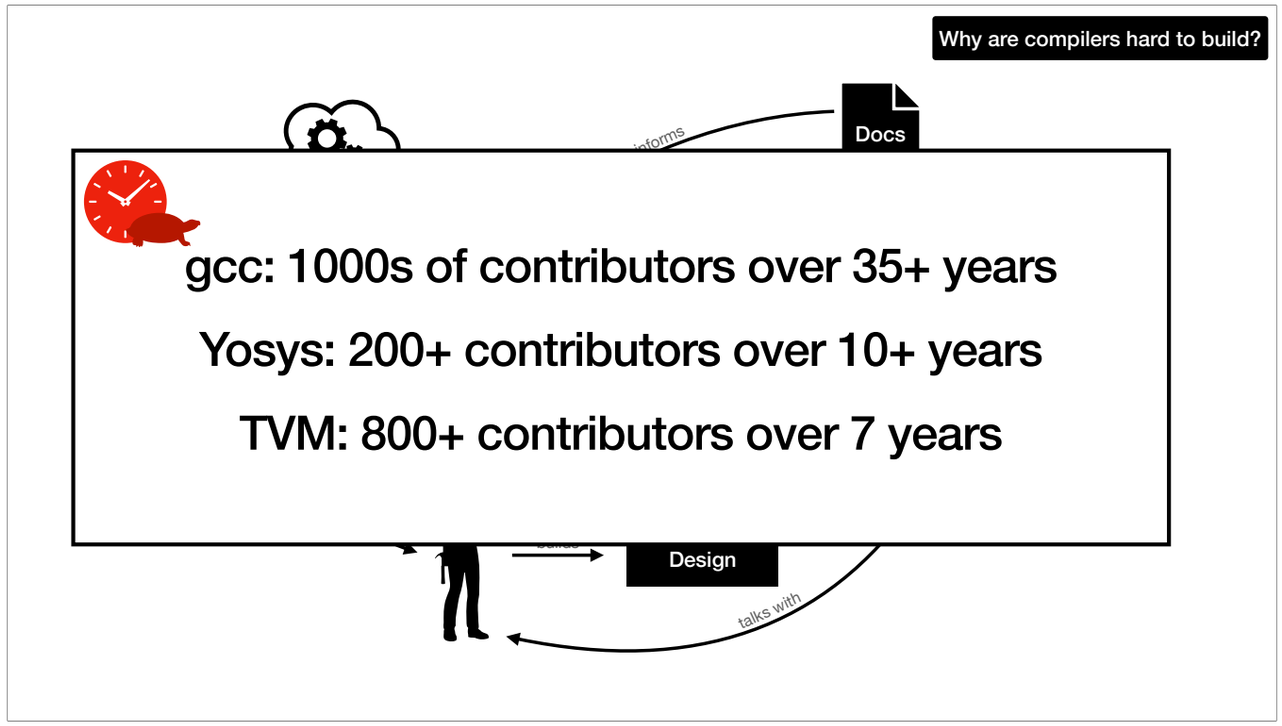

Why are compilers hard to build?
Docs
Programs
Formal Verification Model
builds
informs
informs
compiled by
verifies
Compiler
builds
models
informs
onto
informs
talks with
builds
Hardware Design
talks with


Why are compilers hard to build?
Docs
informs
Programs
Formal Verification Model
builds
compiled by
informs
informs
verifies
Compiler
builds
models
informs
onto
informs
talks with
builds
Hardware Design
talks with

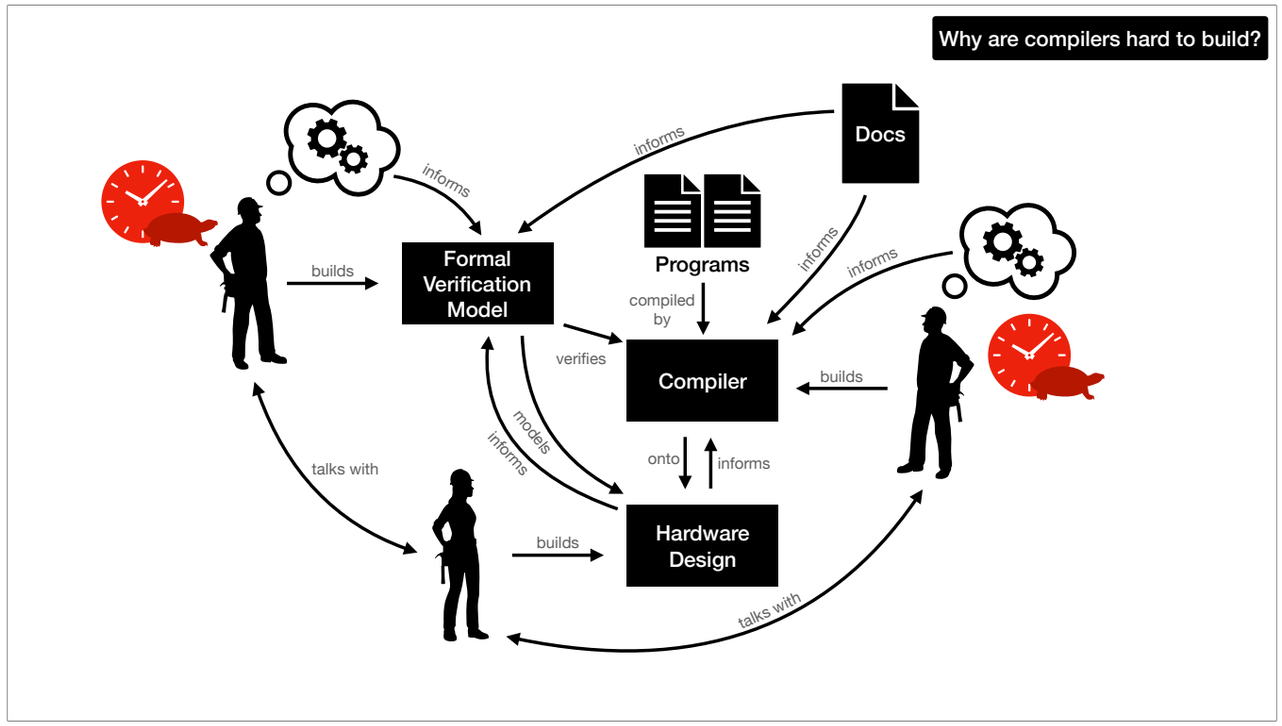

Why are compilers hard to build?
Docs
informs
informs
Programs
informs
informs
builds
Formal Verification Model
compiled by
verifies
Compiler
builds
models
informs
onto
informs
talks with
builds
Hardware Design
talks with


Why are compilers hard to build?
Docs
informs
informs
Programs
informs
informs
builds
Formal Verification Model
compiled by
verifies
Compiler
builds
models
informs
onto
informs
talks with
builds
Hardware Design
talks with

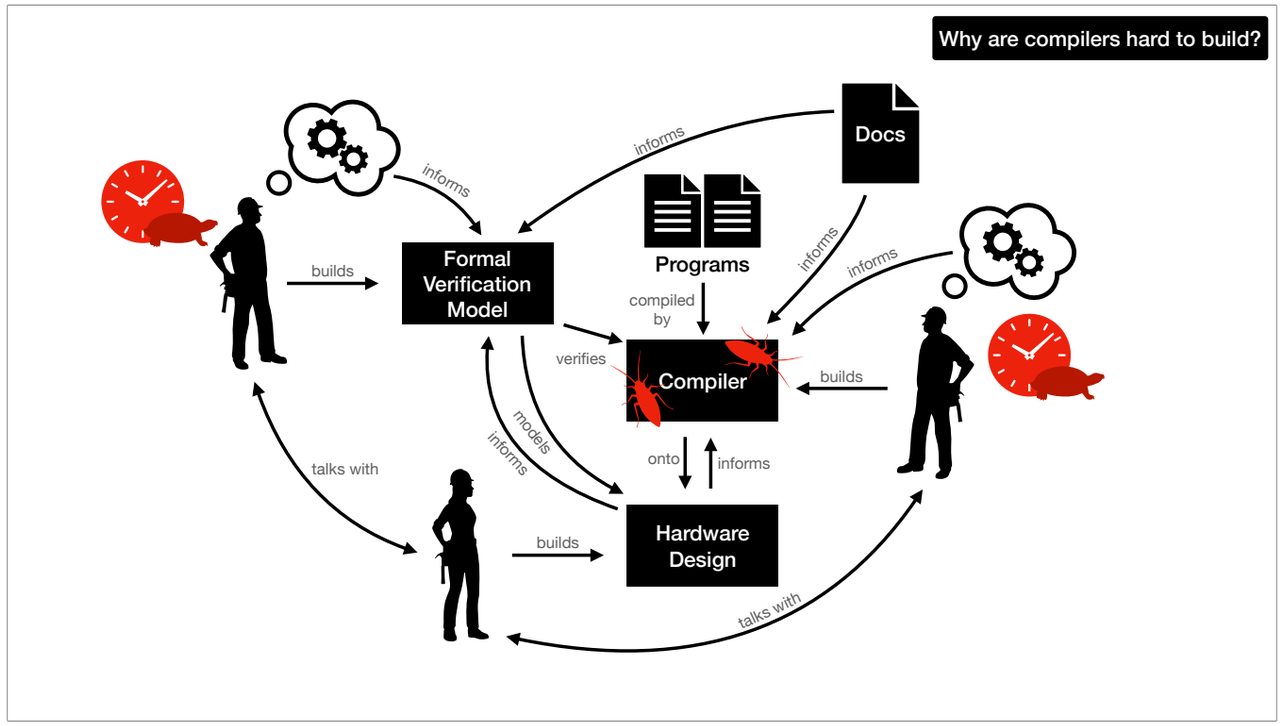

Why are compilers hard to build?
Docs
informs
informs
Programs
informs
informs
builds
Formal Verification Model
compiled by
verifies
Compiler
builds
models
informs
onto
informs
talks with
builds
Hardware Design
talks with

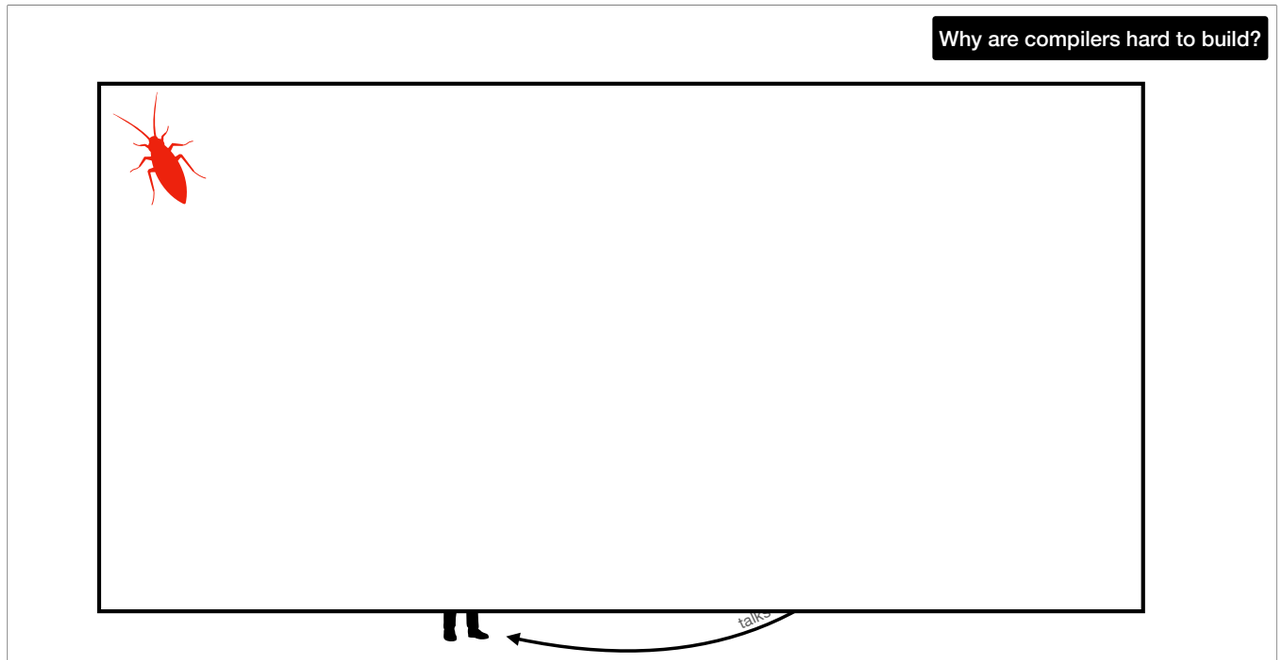


Which is clear once we see the sheer number of individual contributors and time that have gone into major open source compilers like gcc, the hardware synthesis tool Yosys, and the deep learning compiler TVM.



Furthermore, building a compiler is a [build] bug-prone process. In fact,


Why are compilers hard to build?
Docs
informs
informs
Programs
informs
informs
builds
Formal Verification Model
compiled by
verifies
Compiler
builds
models
informs
onto
informs
talks with
builds
Hardware Design
talks with



You can build a strong research career centered on finding and fixing bugs in these large, open-source compilers.

Why are compilers hard to build?

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

(Csmith)

Finds bugs in gcc

Finding and Understanding Bugs in FPGA Synthesis Tools

Yann Herklotz
yann.herklotz15@imperial.ac.uk
Imperial College London
London, UK

John Wickerson
j.wickerson@imperial.ac.uk
Imperial College London
London, UK

(Verismith)

Finds bugs in Yosys

A Comprehensive Study of Deep Learning Compiler Bugs

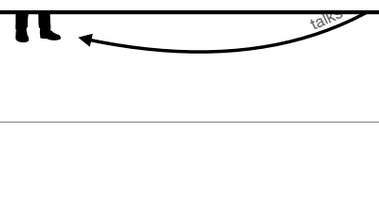
Qingchao Shen
College of Intelligence and Computing, Tianjin University
School of New Media and Communication, Tianjin University
China
qingchao@tju.edu.cn

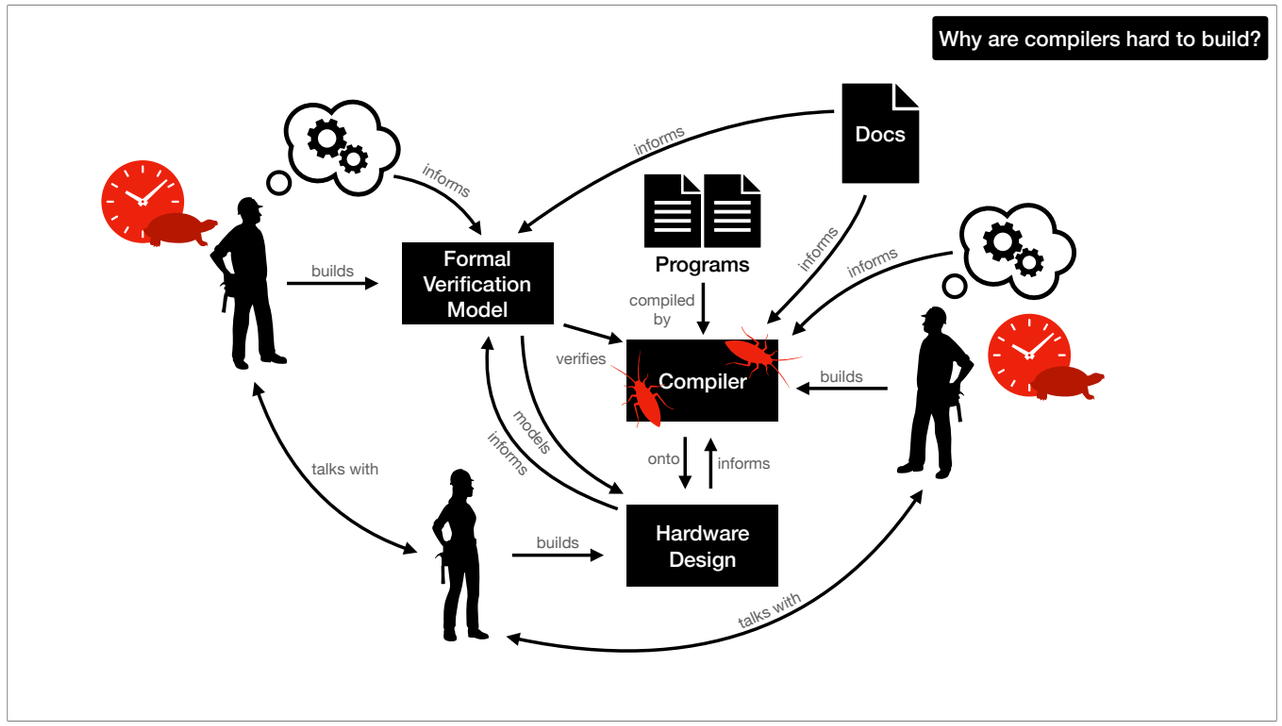
Haoyang Ma
College of Intelligence and Computing, Tianjin University
China
haoyang_9804@tju.edu.cn

Junjie Chen*
College of Intelligence and Computing, Tianjin University
China
junjiechen@tju.edu.cn

(et al.)

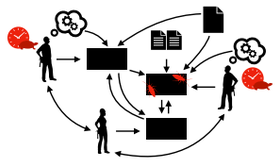
Finds bugs in TVM





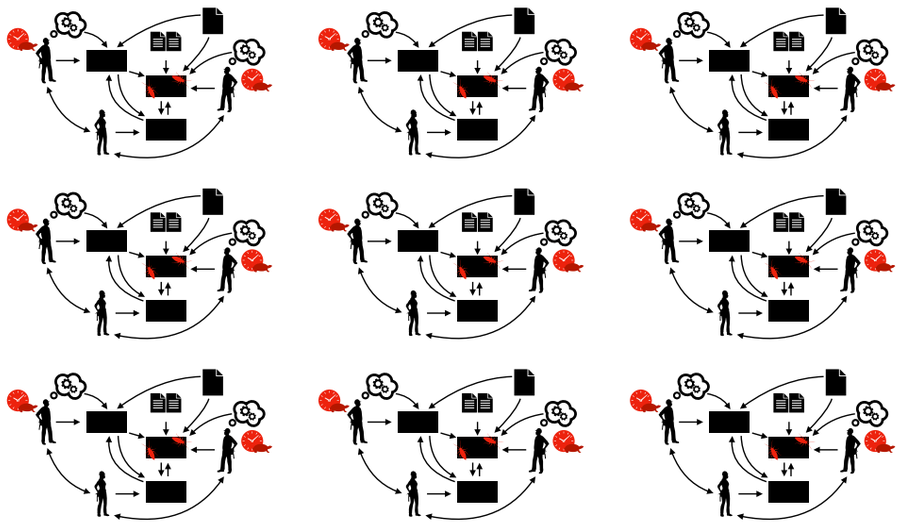
And to make matters worse, these costs are multiplicative. That is,

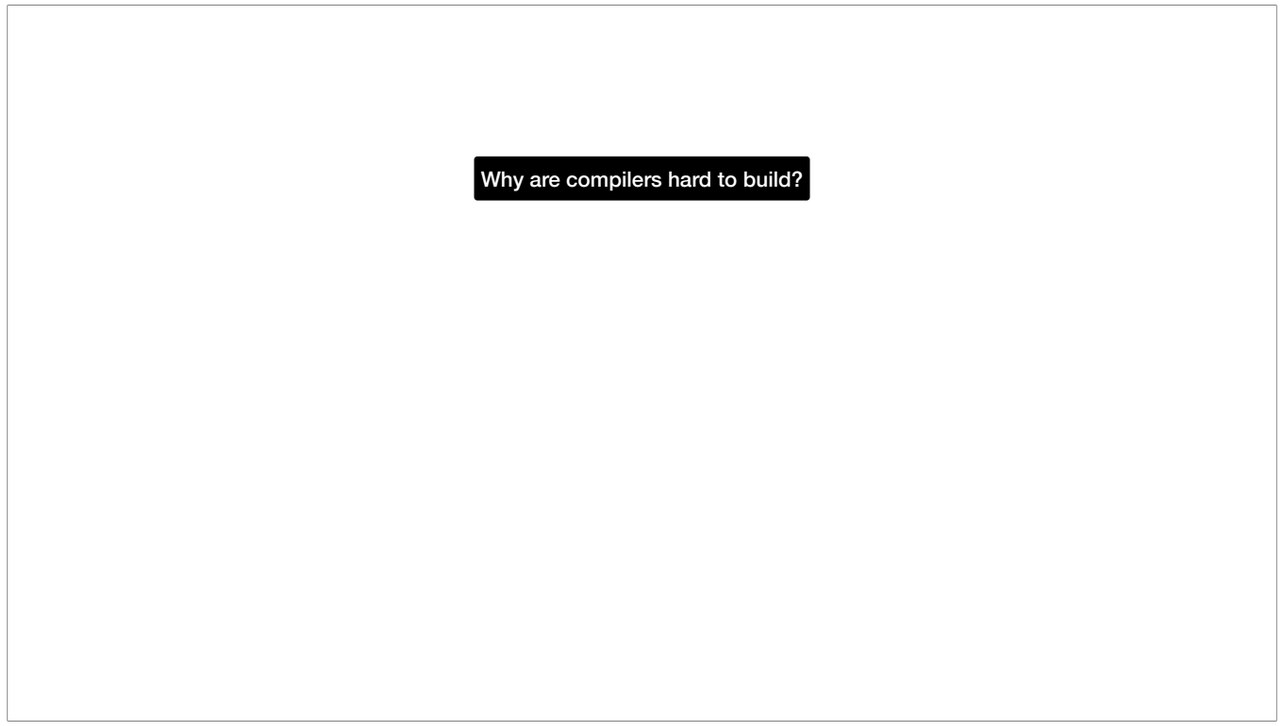
Why are compilers hard to build?



…for each new piece of hardware,
[build] the entire process needs to be repeated to build a new compiler. Though there exists compiler frameworks such as LLVM and MLIR which lessen the burden on compiler engineers, the process still requires significant effort and expertise.

Why are compilers hard to build?





So we asked the question …

What we've seen is that
[build] …
And, importantly,
[build] …

A natural question after all of this is
[build] …
It might sound optimistic given how much effort it is to build a compiler, but let's at least entertain the possibility.

Why are compilers hard to build?

Building a compiler requires significant engineering effort and induces numerous bugs.

Why are compilers hard to build?

Building a compiler requires significant engineering effort and induces numerous bugs.

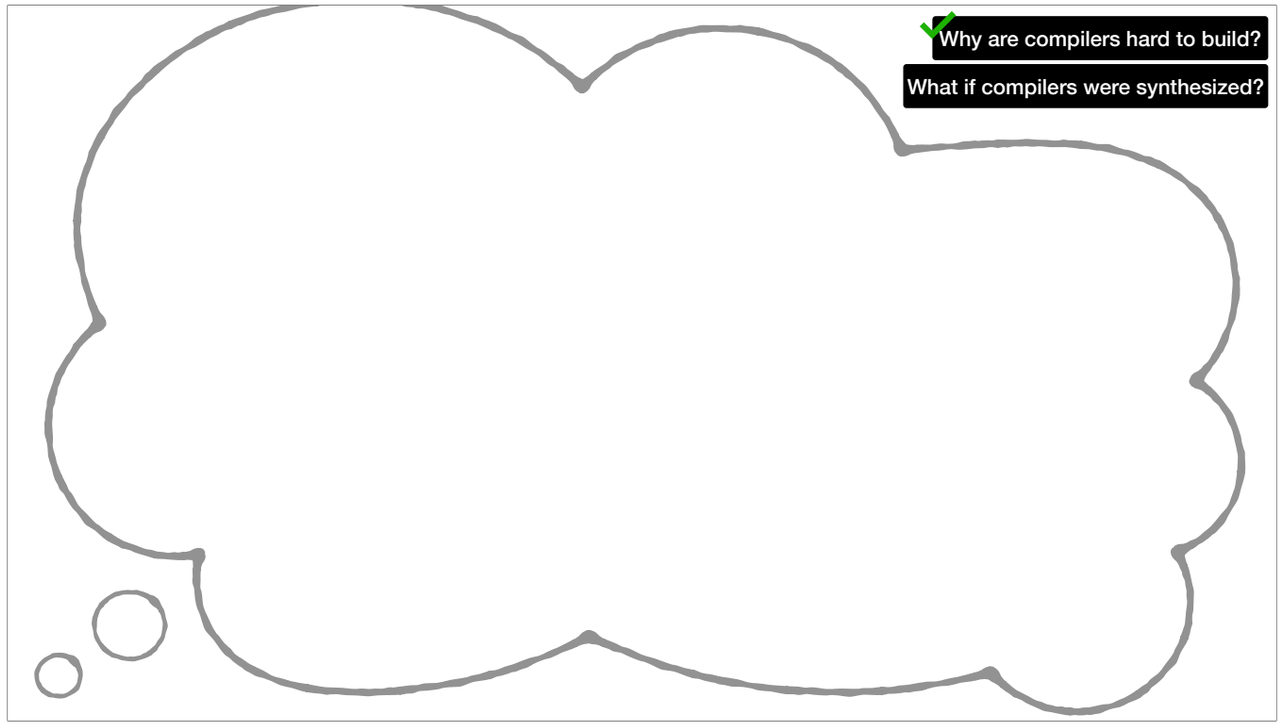
Those costs are multiplied with every new hardware design.

Why are compilers hard to build?

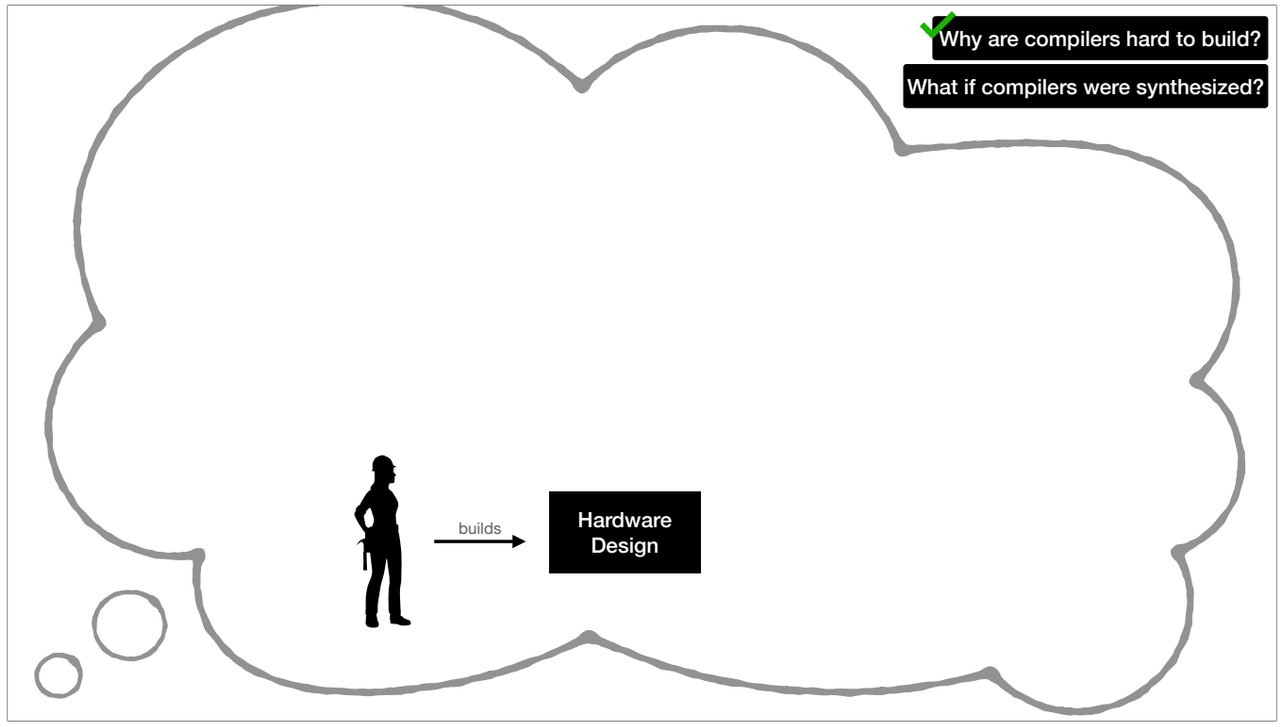
Building a compiler requires significant engineering effort and induces numerous bugs.

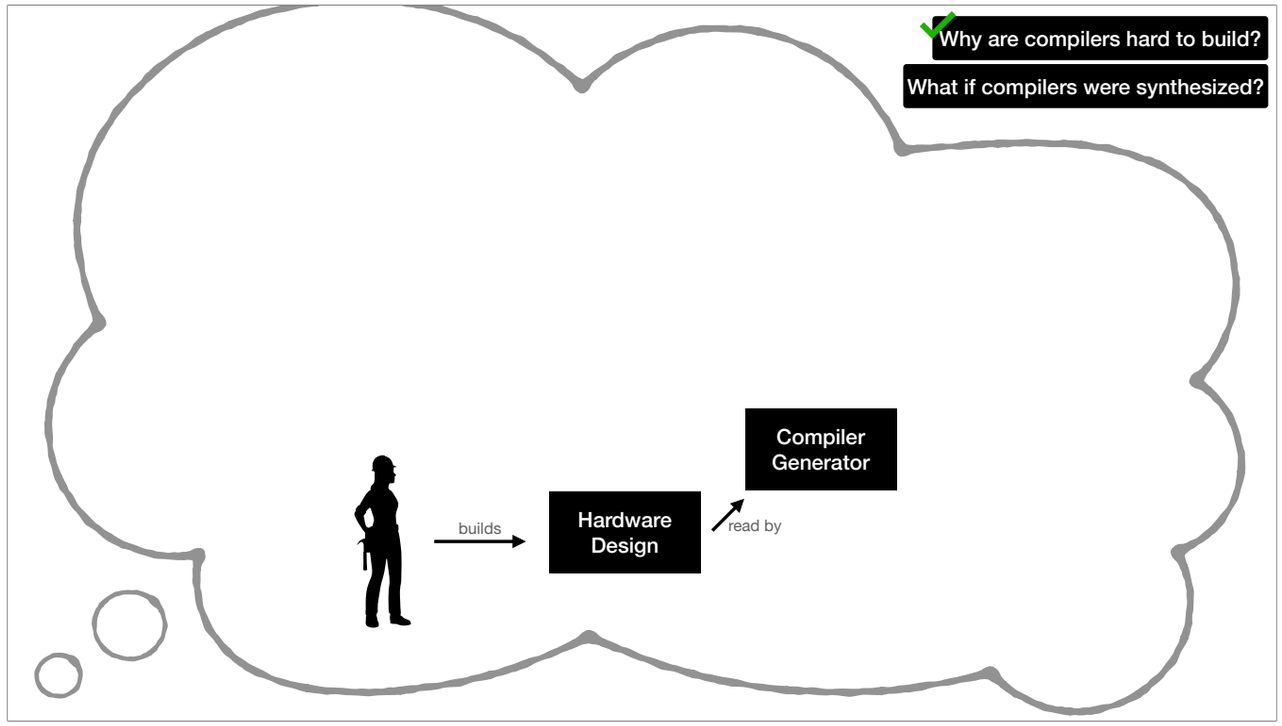
Those costs are multiplied with every new hardware design.

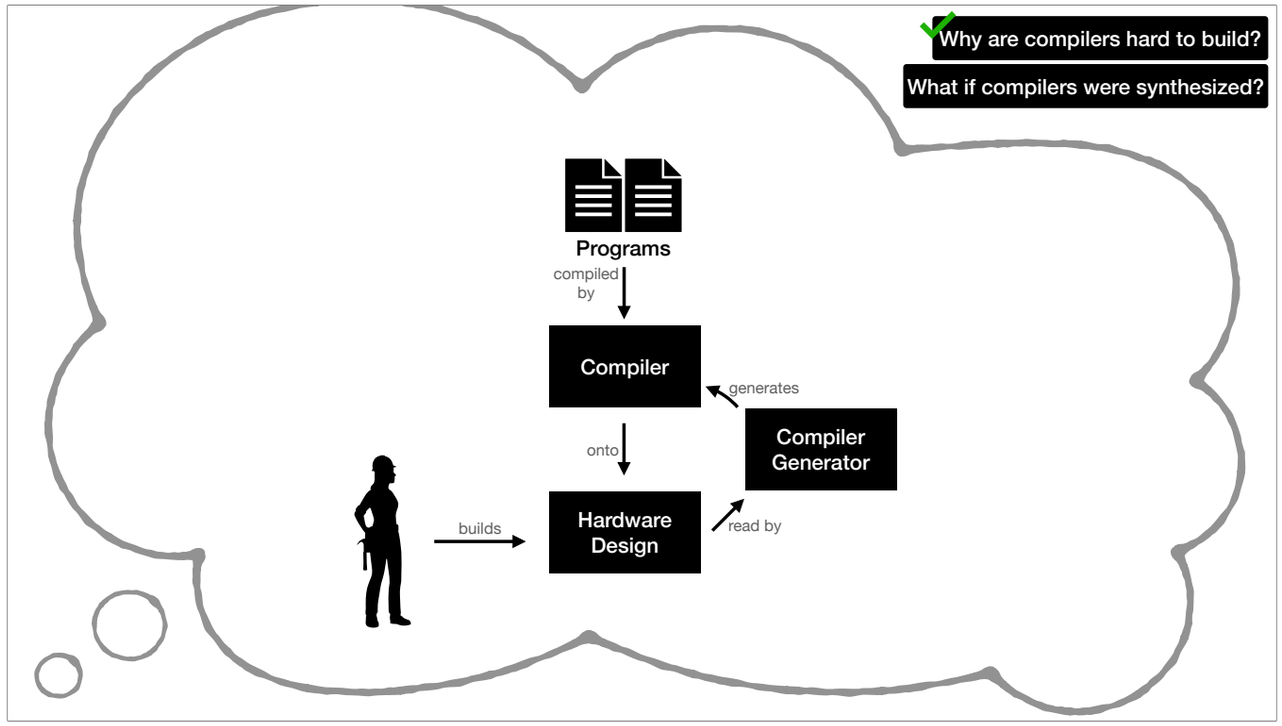
What if compilers were *synthesized?*
(i.e., automatically generated?)

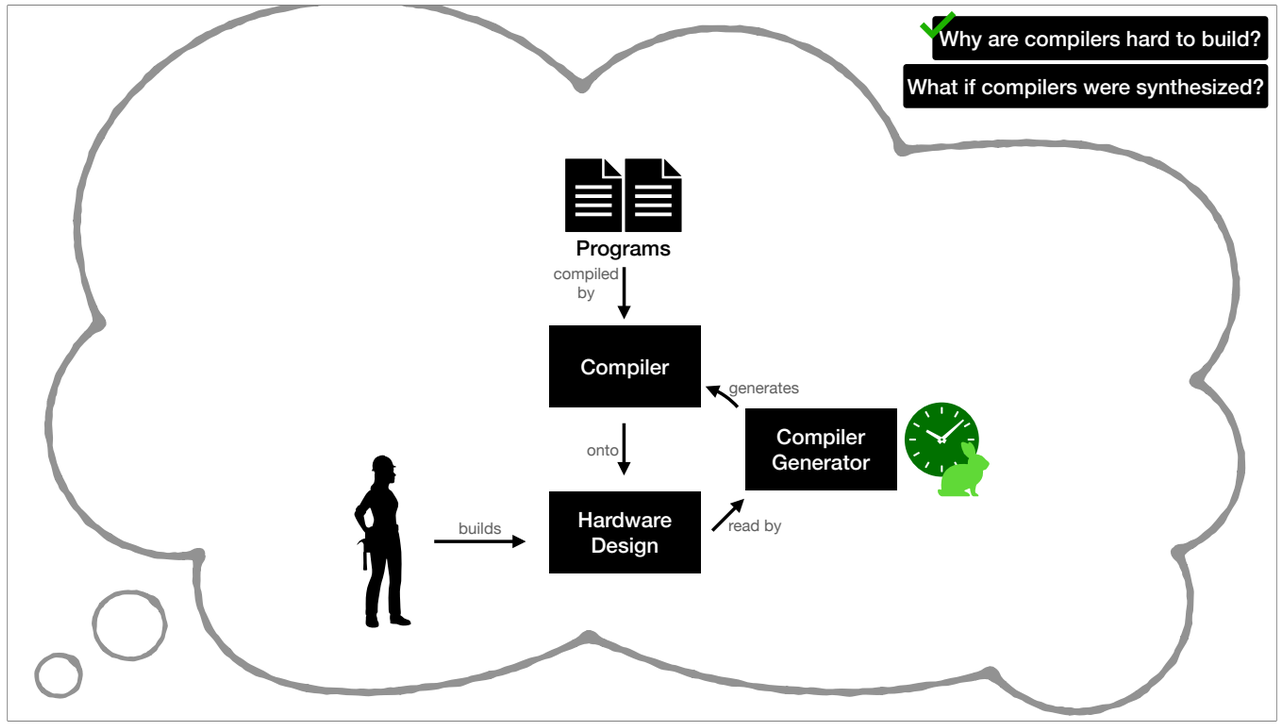


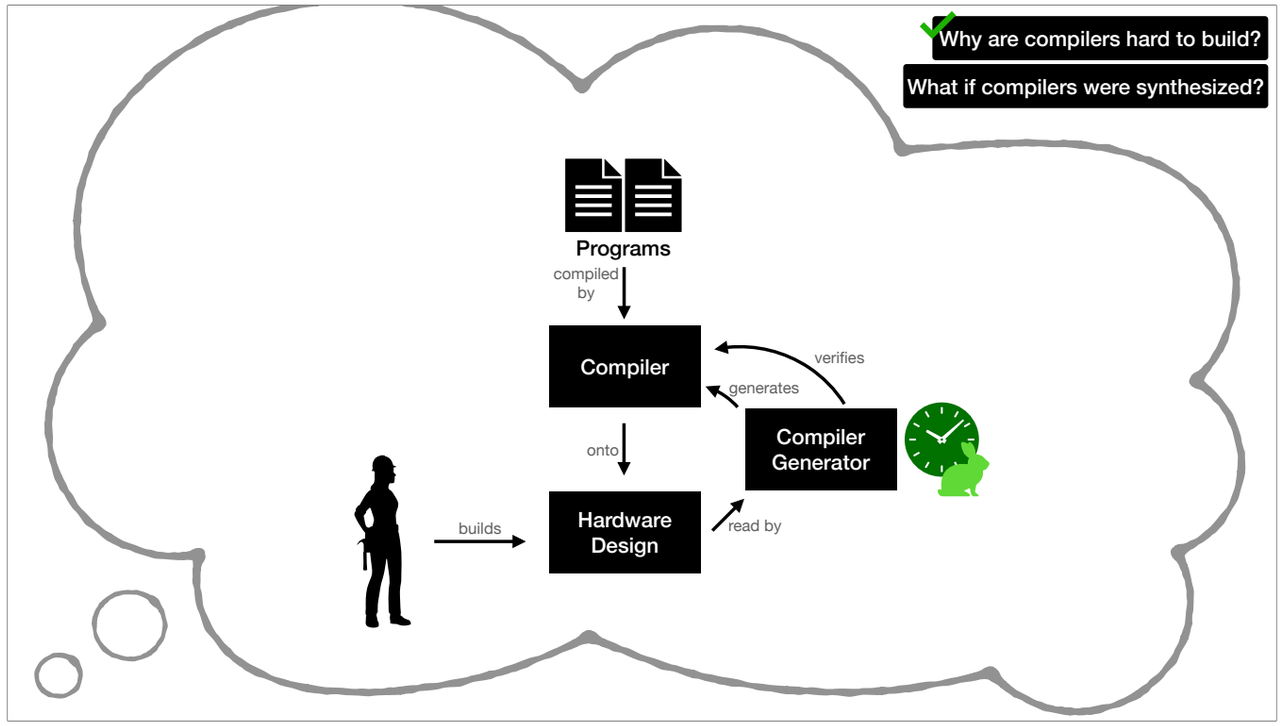
In the ideal case, our
[build] hardware designer still builds their hardware design. But now, that design is read by
[build] a compiler generator, which
[build] generates a compiler directly from their hardware implementation.
Automating compiler construction would save an immense amount of
[build] engineering effort and time.
In addition, depending on how the compiler generator is built, the generator could
[build] verify the compiler as it’s being generated, producing
[build] a bug-free compiler, saving verification time and effort as well.

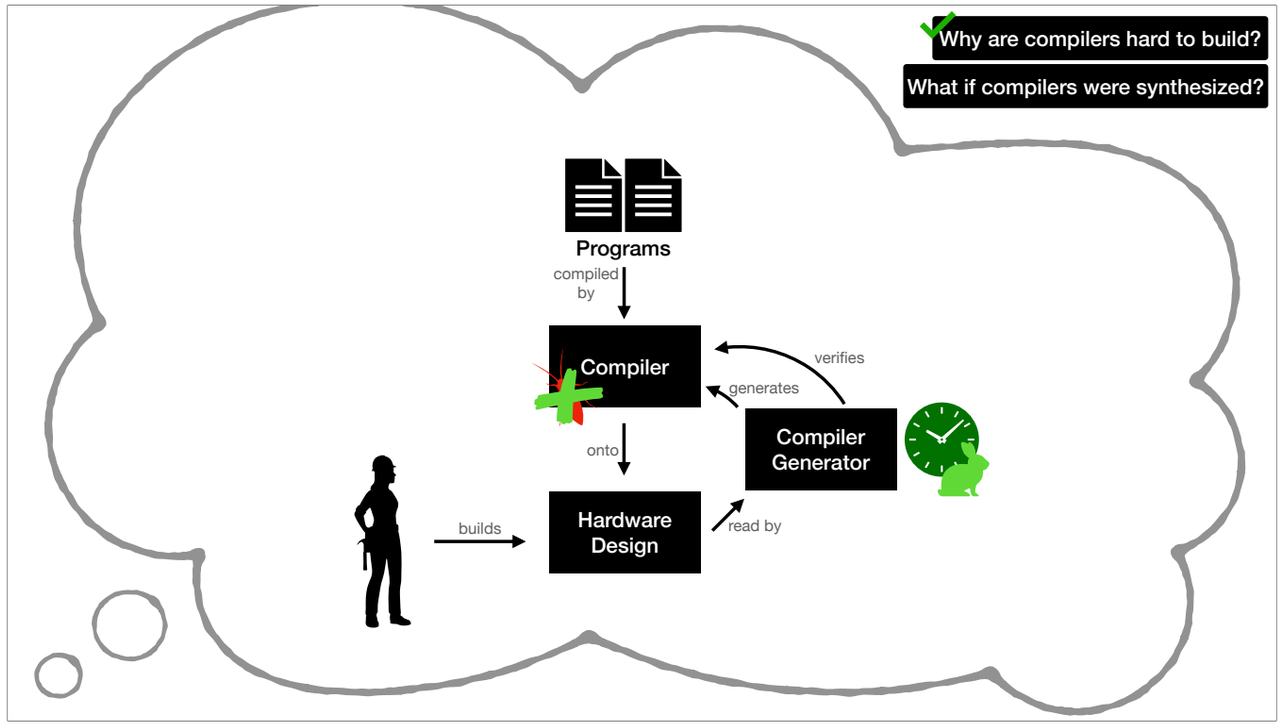

Why are compilers hard to build?
What if compilers were synthesized?
builds
Hardware Design

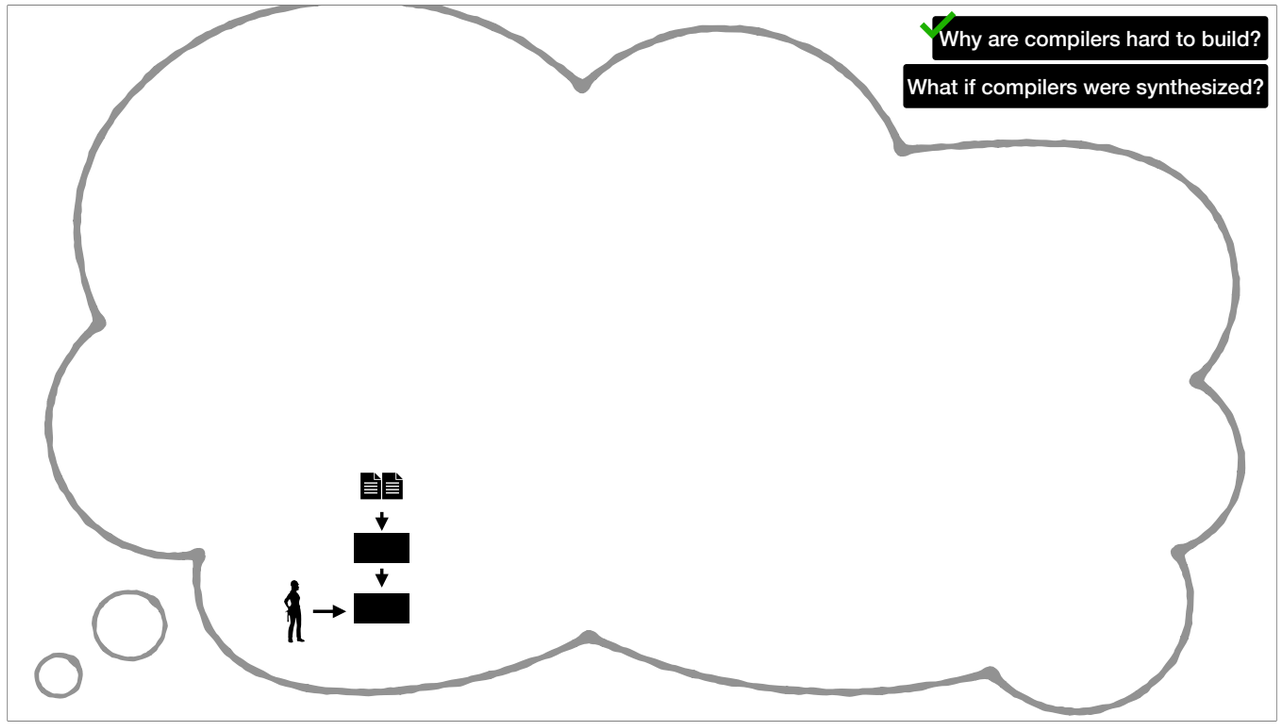

Why are compilers hard to build?
What if compilers were synthesized?
Compiler Generator
builds
Hardware Design
read by


Why are compilers hard to build?
What if compilers were synthesized?
Programs
compiled by
Compiler
generates
Compiler Generator
onto
builds
Hardware Design
read by

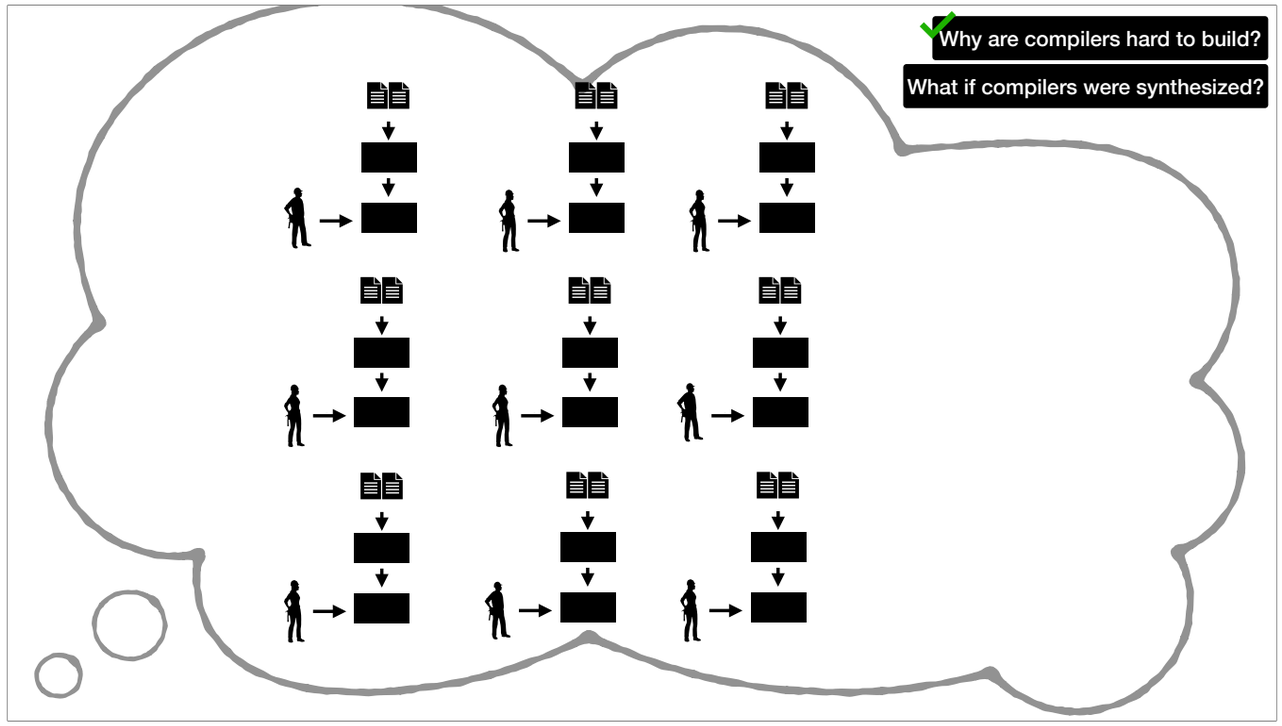

Why are compilers hard to build?
What if compilers were synthesized?
Programs
compiled by
Compiler
generates
Compiler Generator
onto
Hardware Design
read by
builds

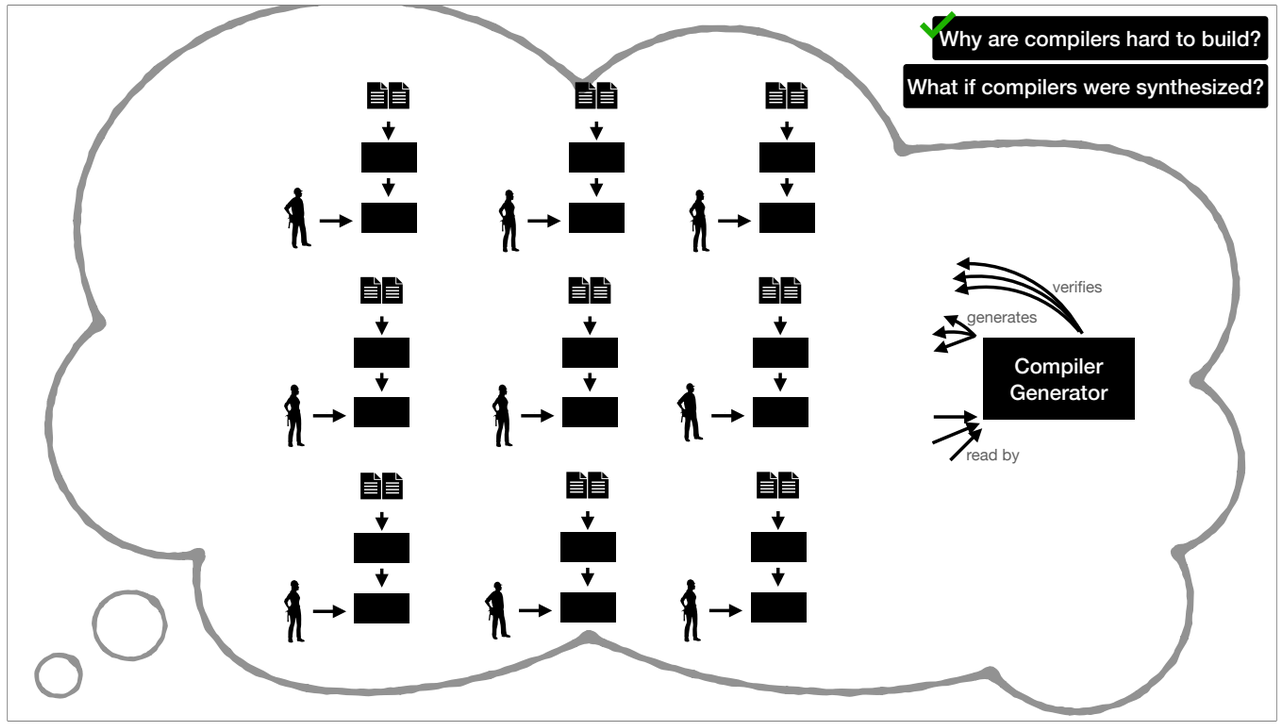

Why are compilers hard to build?
What if compilers were synthesized?
Programs
compiled by
Compiler
verifies
generates
Compiler Generator
onto
builds
Hardware Design
read by

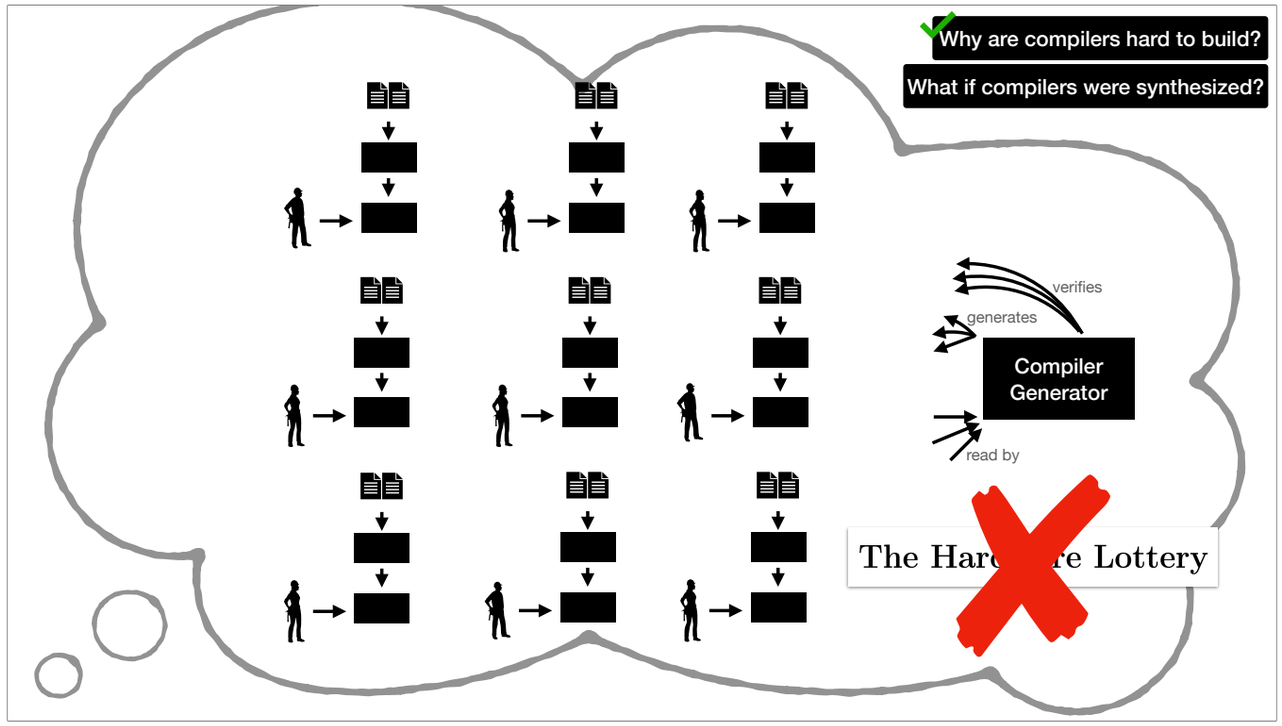

Why are compilers hard to build?
What if compilers were synthesized?
Programs
compiled by
Compiler
verifies
generates
Compiler Generator
onto
builds
Hardware Design
read by



But best of all, this approach can scale
[build] as every new hardware design can reuse
[build] the same compiler generator, removing that multiplicative factor on effort and bugs that exists today.
In this ideal world,
[build] there is no hardware lottery, at least not because of compilers.


Why are compilers hard to build?
What if compilers were synthesized?


Why are compilers hard to build?
What if compilers were synthesized?
verifies
generates
Compiler Generator
read by


Why are compilers hard to build?
What if compilers were synthesized?
verifies
generates
Compiler Generator
read by
The Hard re Lottery

So we see that
[build] …
[build] …

Why are compilers hard to build? ✓

What if compilers were synthesized?

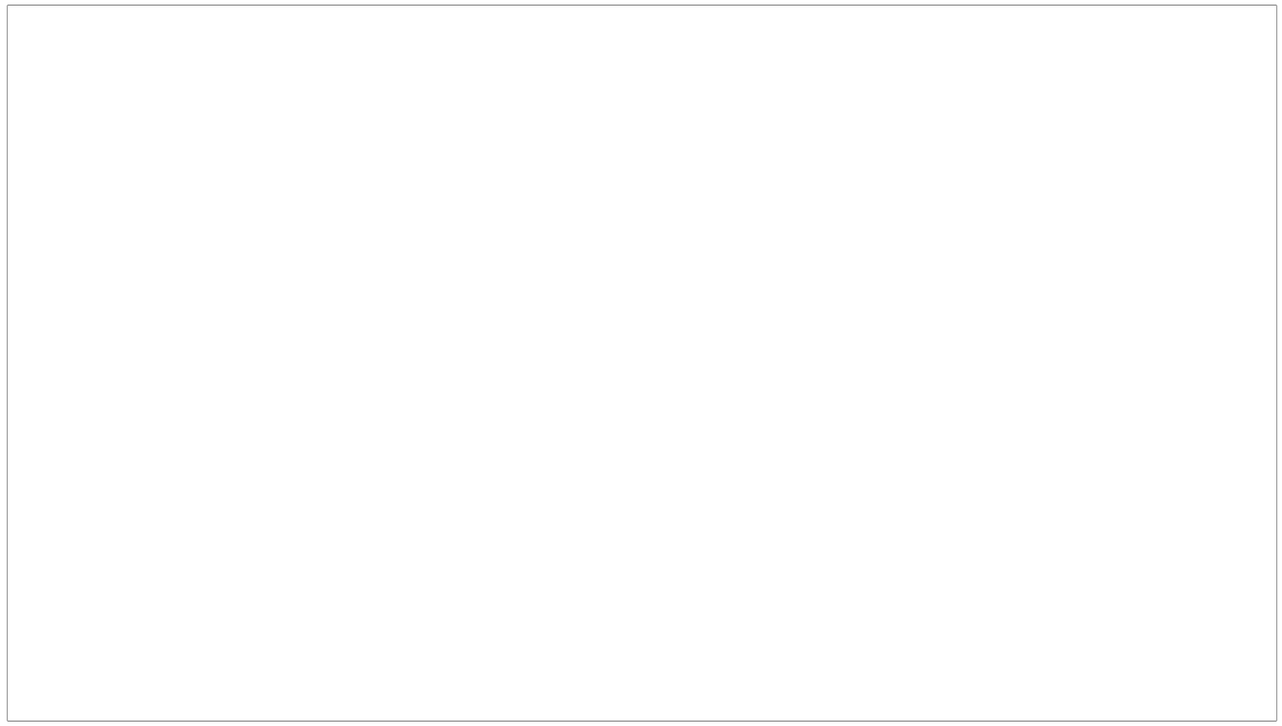
Automatically generating compilers can reduce engineering effort and eliminate bugs.

Why are compilers hard to build? ✓

What if compilers were synthesized?

Automatically generating compilers can reduce engineering effort and eliminate bugs.

Furthermore, the approach scales with new hardware designs, thus fighting against the hardware lottery!

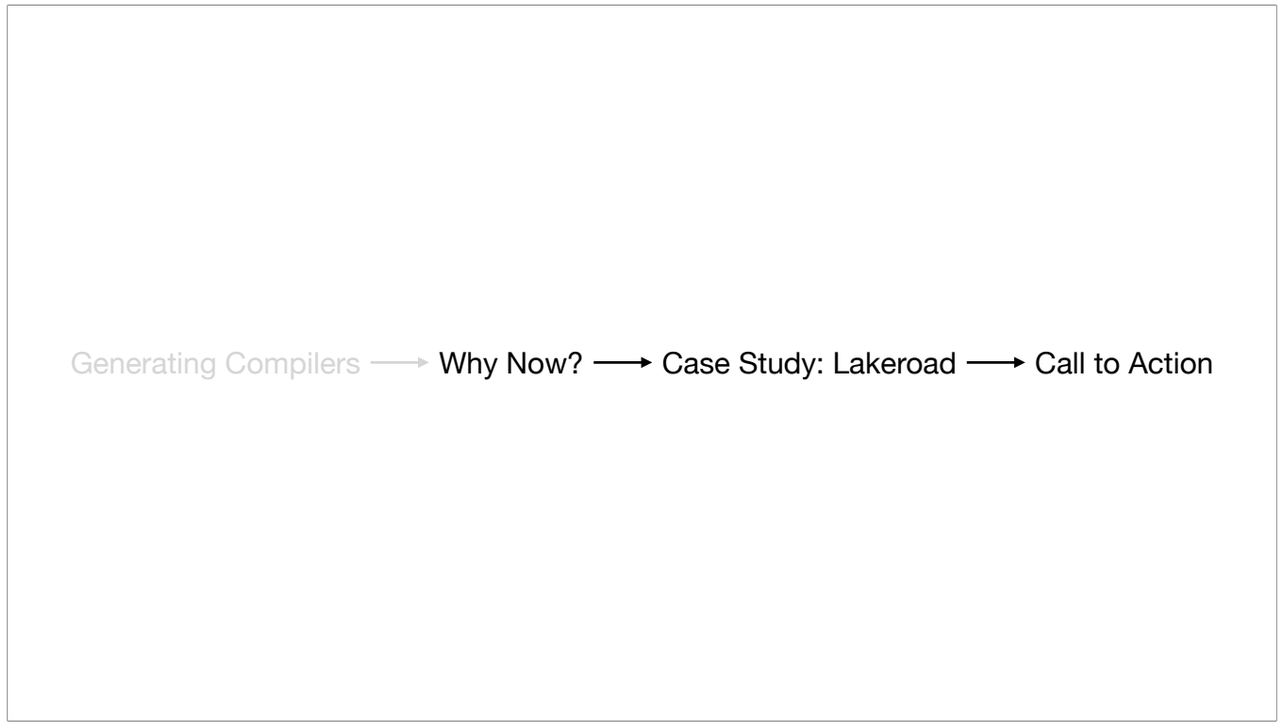


With that, I will now introduce the thesis statement of this talk, in which I claim [build] …

Compilers should be generated from formal models of hardware.

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.



Here’s our roadmap for the rest of the talk.

Let’s talk about why now is the time to do this research.

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

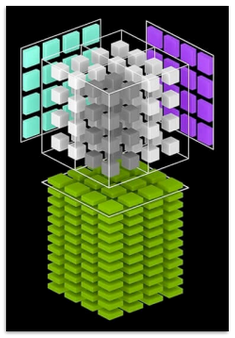
We claim that, with the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make automatic generation of compilers a reality.

Compilers should be generated from formal models of hardware.

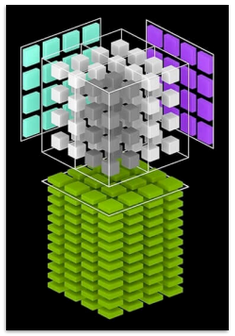
With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

First, let’s talk about what we mean by the growing diversity of hardware.

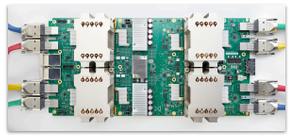
As soon as I mention the diversity of hardware, I'm sure that the first thing that pops into peoples' minds is hardware for machine learning, such as
[build] GPUs and
[build] custom machine learning ASICs.
Yet even within
[build] processors like Apple's A16, we're seeing the addition of specialized accelerators like GPUs and Neural Engines.
Consider also platforms like
[build] Xilinx's Zynq chip, which includes both an ARM CPU and a reconfigurable FPGA, making quite an interesting target for compilers.
Lastly, far from the realm of silicon-based computing, people have begun computing using things like
[build] DNA strand displacement or
[build] metamaterials.
Though these are far from what we would normally consider "hardware", they require compilers nonetheless!
Given the dizzying array of hardware available today, it's clear that
[build] hardware is growing more diverse, and that compilers for new hardware are desperately needed.
This diversity can be intimidating:
[build] how could we possibly generate compilers for all of this hardware?
But the explosion of new hardware platforms actually works in our favor,
[build] because as hardware diversifies, it gets more specialized, and thus, potentially easier to target with automated methods.



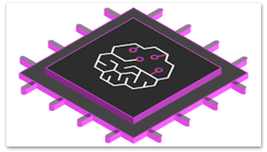
NVIDIA Tensor Cores



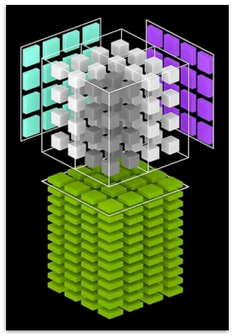
NVIDIA Tensor Cores



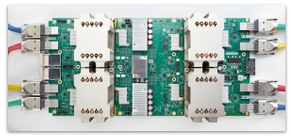
Google TPU



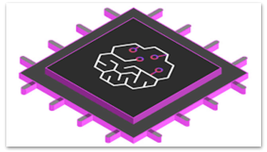
AWS Inferentia



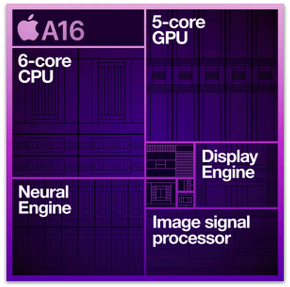
NVIDIA Tensor Cores



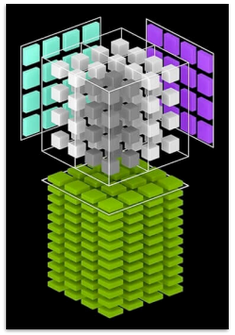
Google TPU



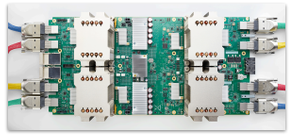
AWS Inferentia



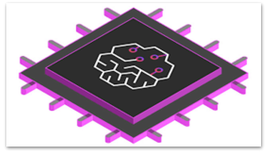
Apple A16



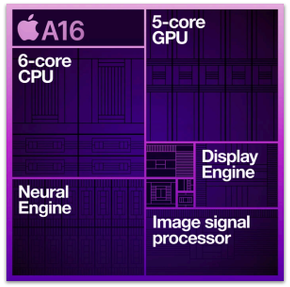
NVIDIA Tensor Cores



Google TPU

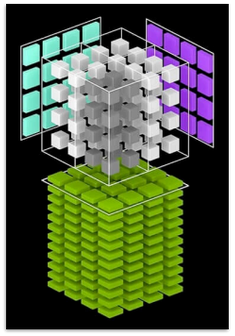


AWS Inferentia

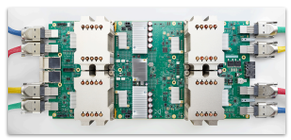


Apple A16

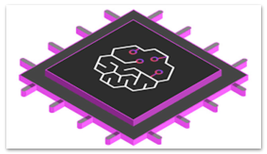
Xilinx Zynq



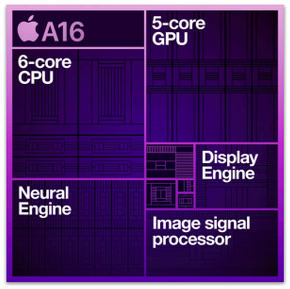
NVIDIA Tensor Cores



Google TPU

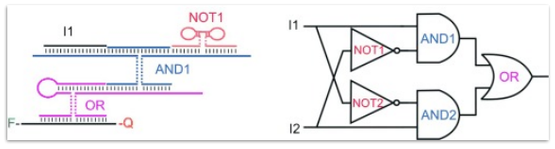


AWS Inferentia

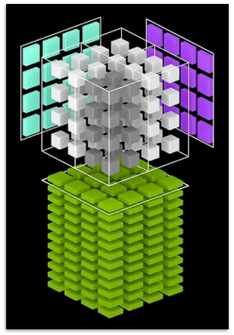


Apple A16

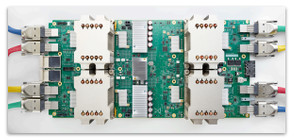
Xilinx Zynq



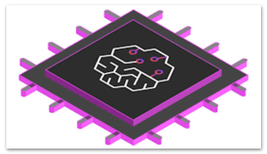
Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*



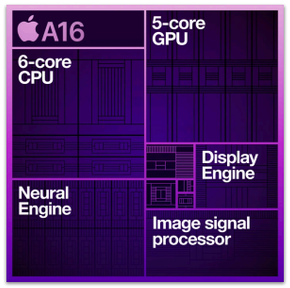
NVIDIA Tensor Cores



Google TPU

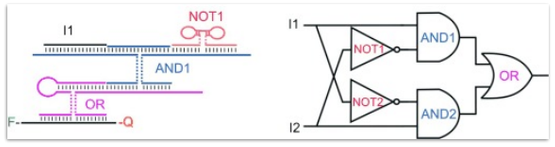


AWS Inferentia

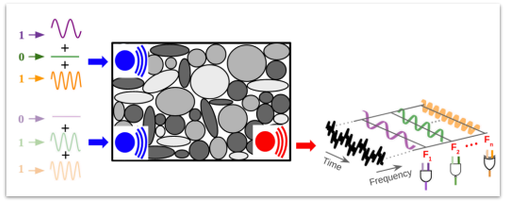


Apple A16

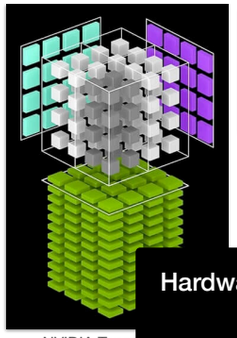
Xilinx Zynq



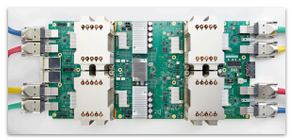
Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*



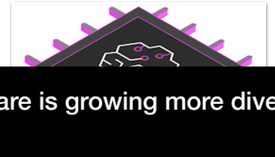
Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*

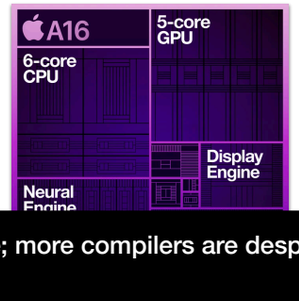


NVIDIA Tens

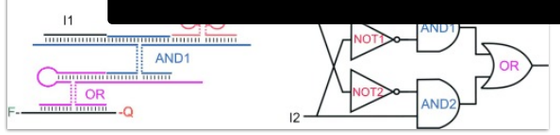


Google TPU

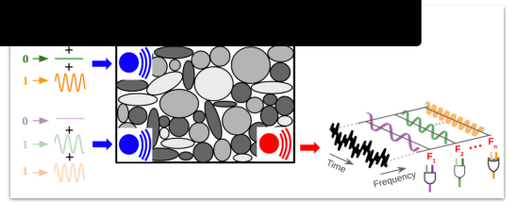




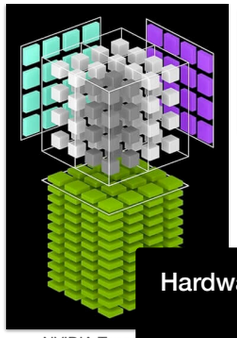
Hardware is growing more diverse; more compilers are desperately needed!

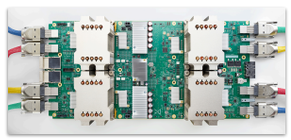


Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*

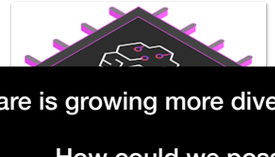


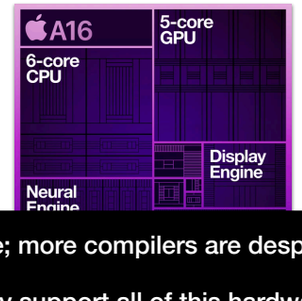
Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*





Google TPU

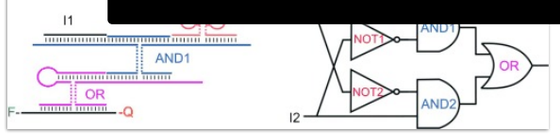




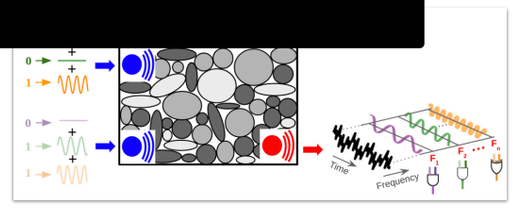
NVIDIA Tens

Hardware is growing more diverse; more compilers are desperately needed!

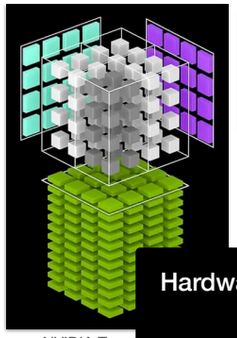
How could we possibly support all of this hardware?

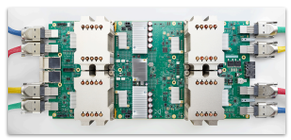


Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*

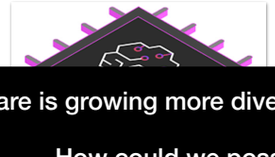


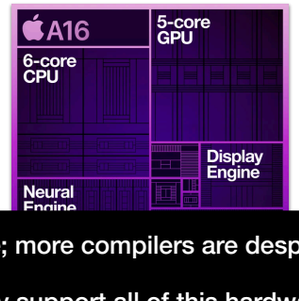
Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*





Google TPU



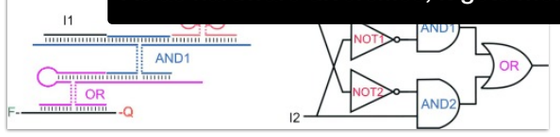


NVIDIA Tens

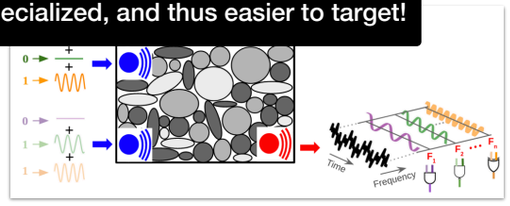
Hardware is growing more diverse; more compilers are desperately needed!

How could we possibly support all of this hardware?

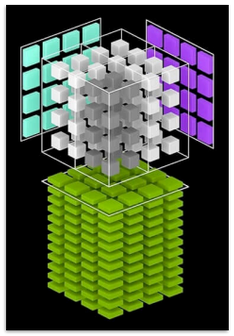
As as hardware diversifies, it gets more specialized, and thus easier to target!



Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*

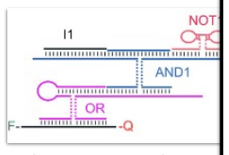


Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*

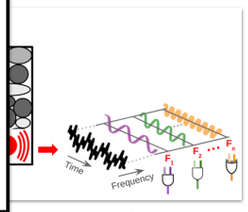


NVIDIA Tensor Cores

Xilinx Zynq

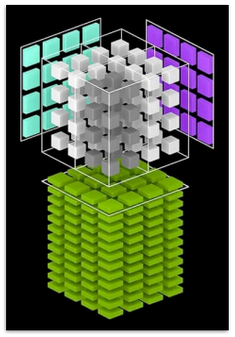


Gerasimova et al. *Connecta*

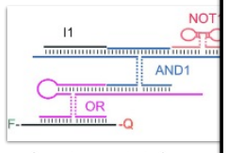


Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*

Previous work on automatically generating compilers largely focuses on
[build] processors.
Generating compilers for processors is
[build] quite a difficult task, as processors are general purpose, and compilers for general purpose processors must handle all of their capabilities.



NVIDIA Tensor Cores



Gerasimova et al. *Connecta*

Describing Instruction Set Processors Using nML

A. Fauth[1] J. Van Praet[2] M. Freericks[1]

[1]Institut für Technische Informatik
Tech. Univ. Berlin, Franklinstr. 28/29
D–10587 Berlin, Germany

[2]IMEC
Kapeldreef 75
B–3001 Leuven, Belgium

1995

Retargetable Generation of Code Selectors from HDL Processor Models

Rainer Leupers, Peter Marwedel
University of Dortmund, Dept. of Computer Science 12, 44221 Dortmund, Germany
email: leupers|marwedel@ls12.informatik.uni-dortmund.de

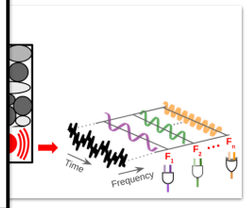
1997

Automatic Tool Generation from Structural Processor Descriptions

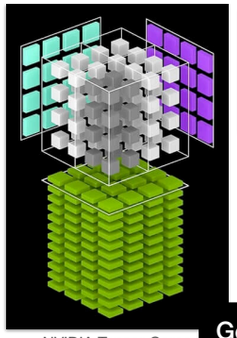
Florian Brandner

2009

Xilinx Zynq



Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*



NVIDIA Tensor Cores

Describing Instruction Set Processors Using nML

A. Fauth[1] J. Van Praet[2] M. Freericks[1]

[1]Institut für Technische Informatik
Tech. Univ. Berlin, Franklinstr. 28/29
D–10587 Berlin, Germany

[2]IMEC
Kapeldreef 75
B–3001 Leuven, Belgium

1995

Retargetable Generation of Code Selectors from HDL Processor Models

University of Dortmund, Dept. of Computer Science 12, 44221 Dortmund, Germany
email: leupers|marwedel@ls12.informatik.uni-dortmund.de

1997

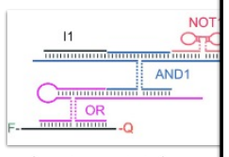
Automatic Tool Generation from Structural Processor Descriptions

Florian Brandner

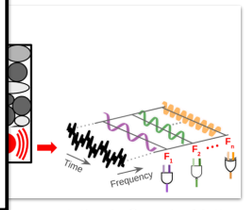
2009

Generating compilers for general-purpose hardware is difficult.

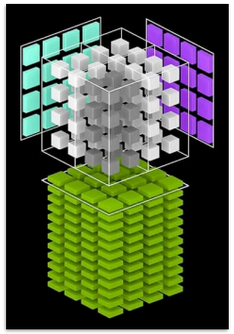
Xilinx Zynq



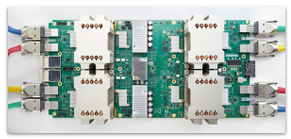
Gerasimova et al. *Connecta*



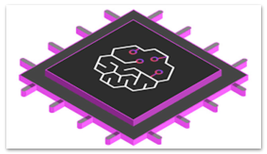
Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*



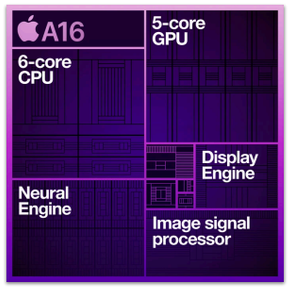
NVIDIA Tensor Cores



Google TPU

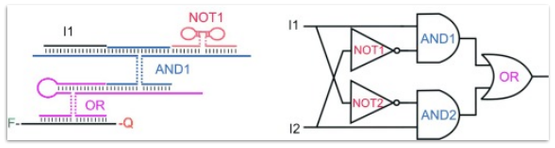


AWS Inferentia

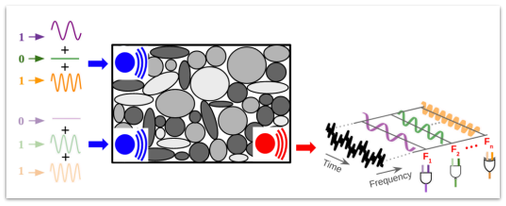


Apple A16

Xilinx Zynq



Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*

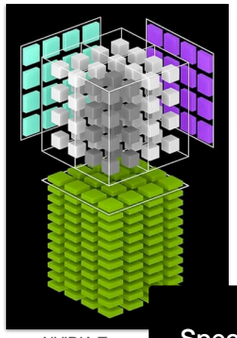


Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*

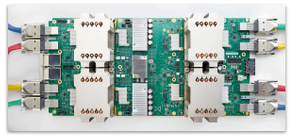
On the other hand, all of the new hardware we're talking about is special purpose,

[build] which makes the task of reasoning about hardware's behavior much more feasible for automated methods.

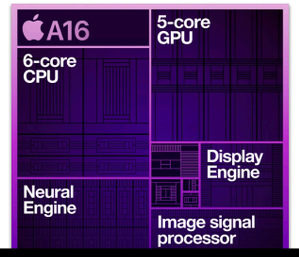
But not only is our hardware more amenable to automated reasoning; our tools for automated reasoning are now powerful enough to take on the task of automated compiler generation.



NVIDIA Tensor

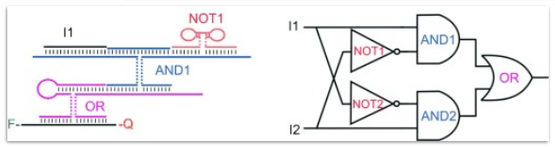


Google TPU

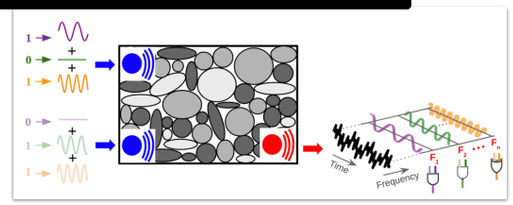


nq

Specialized hardware is easier to target with automated reasoning tools!



Gerasimova et al. *Connectable DNA Logic Gates: OR and XOR Logics.*



Parsa et al. *Universal Mechanical Polycomputation in Granular Matter.*

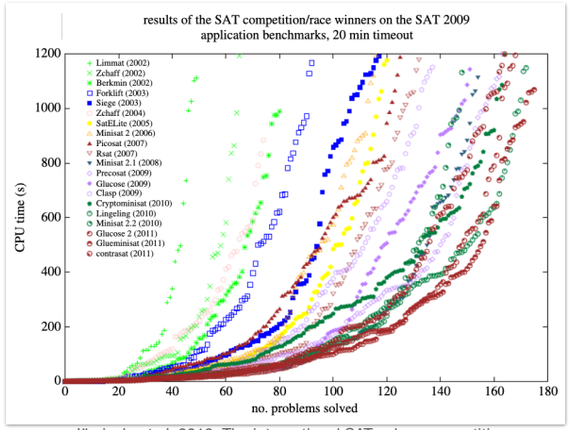
Consider, for example,
[build] SAT and SMT solvers, whose performance have been
[build] steadily increasing,

Or the relatively new technique of
[build] equality saturation, which has already shown great promise for compiler construction,

And of course it wouldn't be a talk in 2023 if I didn't mention
[build] large language models, which are powerful tools for generating hardware code, among other tasks.

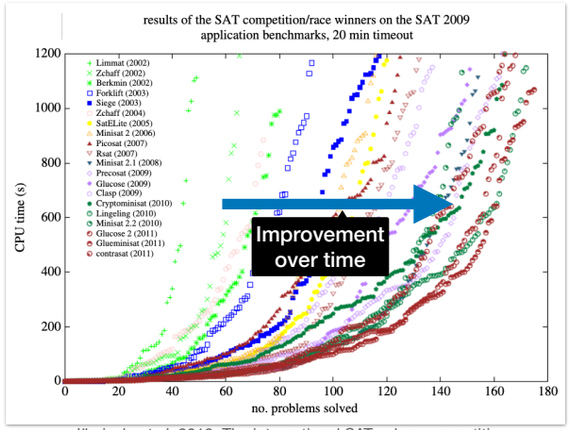
Given that this is only a small selection of the automated reasoning tools available, it's clear that
[build] …

SAT/SMT



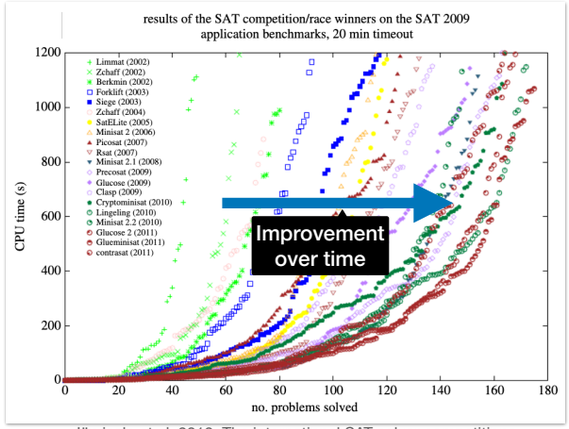
Järvisalo et al. 2012. The international SAT solver competitions.

SAT/SMT



Järvisalo et al. 2012. The international SAT solver competitions.

SAT/SMT



Järvisalo et al. 2012. The international SAT solver competitions.

Equality Saturation

Vectorization for Digital Signal Processors via Equality Saturation

Alexa VanHattum
Cornell University
Ithaca, NY, USA

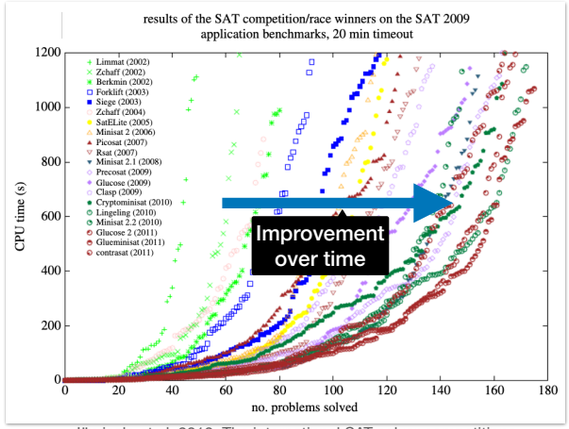
Rachit Nigam
Cornell University
Ithaca, NY, USA

Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA

James Bornholt
The University of Texas at Austin
Austin, TX, USA

Adrian Sampson
Cornell University
Ithaca, NY, USA

SAT/SMT



Järvisalo et al. 2012. The international SAT solver competitions.

Equality Saturation

Vectorization for Digital Signal Processors via Equality Saturation

Alexa VanHattum
Cornell University
Ithaca, NY, USA

Rachit Nigam
Cornell University
Ithaca, NY, USA

Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA

James Bornholt
The University of Texas at Austin
Austin, TX, USA

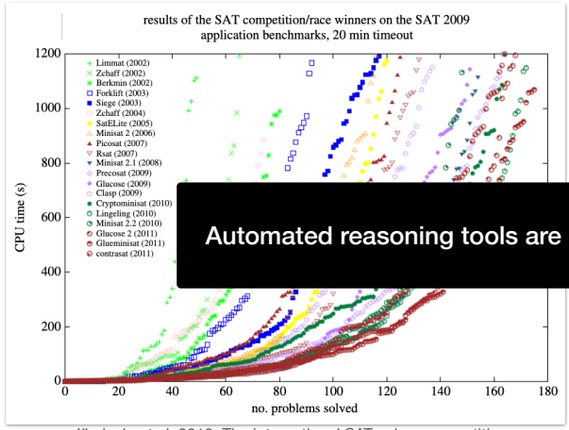
Adrian Sampson
Cornell University
Ithaca, NY, USA

Benchmarking Large Language Models for Automated Verilog RTL Code Generation

Shailja Thakur*, Baleegh Ahmad*, Zhenxing Fan*, Hammond Pearce*,
Benjamin Tan†, Ramesh Karri*, Brendan Dolan-Gavitt*, Siddharth Garg*
*New York University, †University of Calgary

ML/LLMs

SAT/SMT



Järvisalo et al. 2012. The international SAT solver competitions.

Equality Saturation

Vectorization for Digital Signal Processors via Equality Saturation

Alexa VanHattum
Cornell University
Ithaca, NY, USA

Rachit Nigam
Cornell University
Ithaca, NY, USA

Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA

James Bornholt
The University of Texas at Austin

Adrian Sampson
Cornell University

Automated reasoning tools are ready for the task of compiler generation.

ML/LLMs

Models for Automated Verilog RTL Code Generation

Shailja Thakur*, Baleegh Ahmad*, Zhenxing Fan*, Hammond Pearce*,
Benjamin Tan†, Ramesh Karri*, Brendan Dolan-Gavitt*, Siddharth Garg*
*New York University, †University of Calgary

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

So, why is now the time to make the automatic generation of compilers a reality? Well,
[build] …
Furthermore,
[build] …
Finally,
[build] …

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

Hardware is diversifying, and we need new compilers.

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

Hardware is diversifying, and we need new compilers.
Modern targets are more amenable to automated methods.

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

Hardware is diversifying, and we need new compilers.
Modern targets are more amenable to automated methods.
Automated reasoning tools are ready for the task of compiler generation.

Now,

…let’s talk about a concrete example of generating a compiler from hardware models, in a project we call Lakeroad.

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

We keep talking about the growing diversity of hardware platforms. Now, [build] let's look at a concrete example: FPGAs.

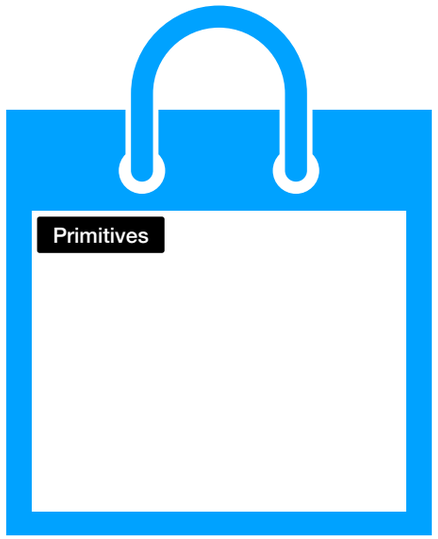
Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

Let's look at a concrete example: FPGAs.

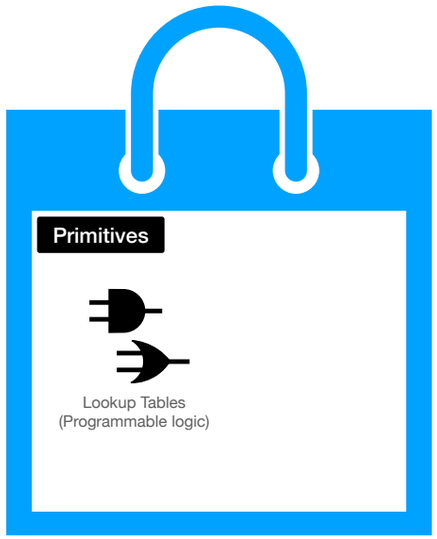
FPGAs are reconfigurable devices that can be used to implement hardware.

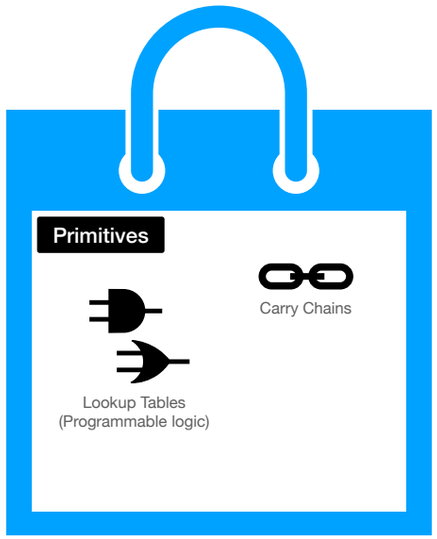
At a high level, though, you can think of an FPGA as being a bag filled with parts, or primitives.

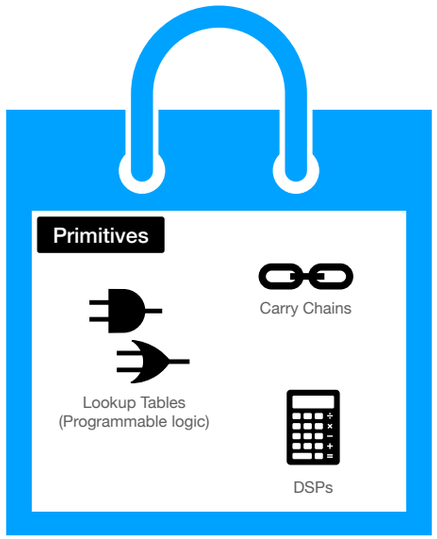


In the past, FPGAs consisted only of
[build] lookup tables, which are primitives that can be configured to implement logic gates.

Over the years, FPGAs have added specialized primitives such as
[build] carry chains to implement fast arithmetic.
One of the most interesting and impactful additions to FPGAs has been the inclusion of
[build] digital signal processors or DSPs, which are small, programmable embedded processors.


Primitives
Lookup Tables
(Programmable logic)


Primitives
Carry Chains
Lookup Tables
(Programmable logic)


Primitives
Carry Chains
Lookup Tables
(Programmable logic)
DSPs

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapi ated reasoning, his a reality.

Even *within* FPGAs, hardware is diversifying.

So we can see that

[build] …
[build] Are new primitives a challenge for FPGA compilers, as our thesis would suggest?

Compilers should be generated from formal models of hardware.

With the growing diversity of hardware and the rapi ated reasoning, his a reality.

Even *within* FPGAs, hardware is diversifying.

Are new primitives a challenge for FPGA compilers?

Are new primitives a challenge for FPGA compilers?

```
((d + a) * b) ^ c
```

To test this, we will attempt to compile a simple hardware design onto a Xilinx FPGA's DSP.

In our case, our simple design takes four inputs and computes this expression in three pipeline stages.

Are new primitives a challenge for FPGA compilers?

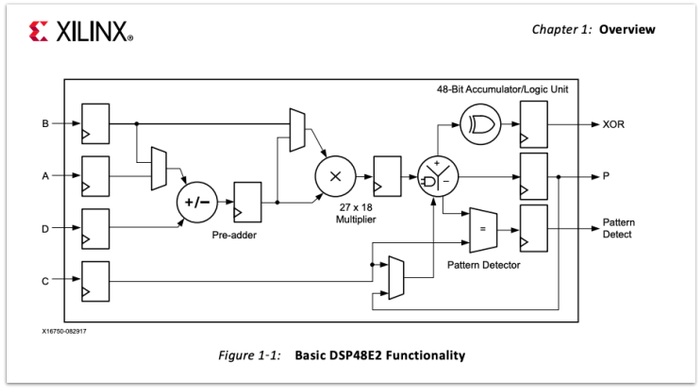


Figure 1-1: **Basic DSP48E2 Functionality**

```
((d + a) * b) ^ c
```

The Xilinx DSP documentation claims that this expression is supported on the DSP. But when we attempt to compile our design using the state of the art compiler for Xilinx FPGAs, we see a surprising result.

Are new primitives a challenge for FPGA compilers?

```
Report Cell Usage:
+------+--------+------+
|      |Cell    |Count |
+------+--------+------+
|1     |DSP48E1 |     2|
|2     |LUT2    |    10|
|3     |SRL16E  |    10|
|4     |FDRE    |    10|
+------+--------+------+
```

We see that the design is in fact compiled onto DSPs; but instead of
[build] using a single DSP as expected, it uses two DSPs and ten look up tables.

Are new primitives a challenge for FPGA compilers?

```
Report Cell Usage:
+------+--------+------+
|      |Cell    |Count |
+------+--------+------+
|1     |DSP48E1 |     2|
|2     |LUT2    |    10|
|3     |SRL16E  |    10|
|4     |FDRE    |    10|
+------+--------+------+
```

Should use only a single DSP!

Are new primitives a challenge for FPGA compilers?

So, are new primitives…

From our brief experiment, it seems like the answer is
[build] yes!
[build] But this is unsurprising…

Are new primitives a challenge for FPGA compilers?

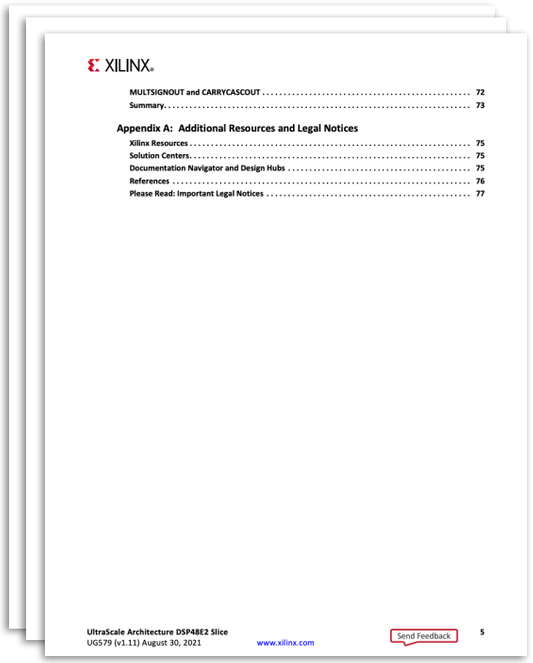
On brief inspection, yes!

Are new primitives a challenge for FPGA compilers?

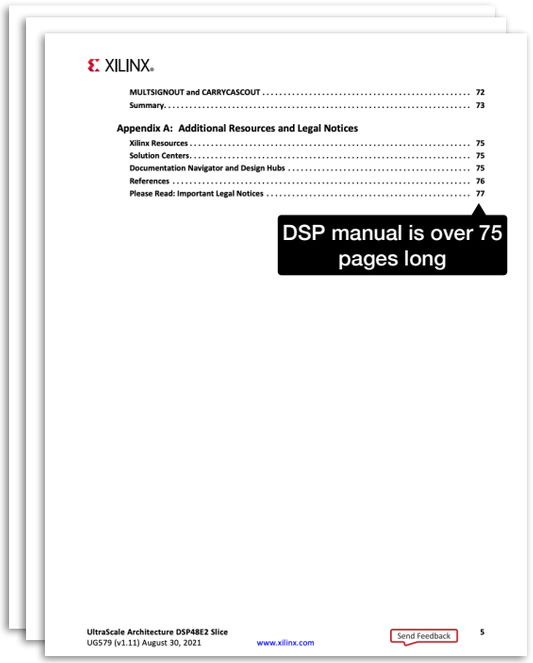
On brief inspection, yes!

But this is unsurprising—DSPs are complicated.

The manual for our Xilinx DSP alone is [build] over 75 pages long.



XILINX®


XILINX
MULTSIGNOUT and CARRYCASCOUT 72
Summary 73
Appendix A: Additional Resources and Legal Notices
Xilinx Resources 75
Solution Centers 75
Documentation Navigator and Design Hubs 75
References 76
Please Read: Important Legal Notices 77
DSP manual is over 75 pages long
UltraScale Architecture DSP48E2 Slice
UG579 (v1.11) August 30, 2021
www.xilinx.com
Send Feedback
5

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  parameter integer ALUMODEREG = 1,
  parameter AMULTSEL = "A",
  parameter integer AREG = 1,
  parameter AUTORESET_PATDET = "NO_RESET",
  parameter AUTORESET_PRIORITY = "RESET",
  parameter A_INPUT = "DIRECT",
  ...
)(
  output [29:0] ACOUT,
  output [17:0] BCOUT,
  output CARRYCASCOUT,
  ...

  input [29:0] A,
  input [29:0] ACIN,
  input [3:0] ALUMODE,
  input [17:0] B,
  input [17:0] BCIN,
  input [47:0] C,
  ...
);
```

Furthermore, instantiating a DSP requires
[build] setting over 100 ports and parameters, while
[build] obeying strict requirements on port and parameter values, described throughout the 75 page manual.

With all of these complex interdependencies on what values are legal for which parameters based on the values of other parameters,
[build] configuring a DSP is starting to sound a lot like writing a program.

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  parameter integer ALUMODEREG = 1,
  parameter AMULTSEL = "A",
  parameter integer AREG = 1,
  parameter AUTORESET_PATDET = "NO_RESET",
  parameter AUTORESET_PRIORITY = "RESET",
  parameter A_INPUT = "DIRECT",
  ...
)(
  output [29:0] ACOUT,
  output [17:0] BCOUT,
  output CARRYCASCOUT,
  ...

  input [29:0] A,
  input [29:0] ACIN,
  input [3:0] ALUMODE,
  input [17:0] B,
  input [17:0] BCIN,
  input [47:0] C,
  ...
);
```

Configuring the DSP requires setting 100+ ports and parameters

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  parameter integer ALUMODEREG = 1,
  parameter AMULTSEL = "A",
  parameter integer AREG = 1,
  parameter AUTORESET_PATDET = "NO_RESET",
  parameter AUTORESET_PRIORITY = "RESET",
  parameter A_INPUT = "DIRECT",
  ...
)(
  output [29:0] ACOUT,
  output [17:0] BCOUT,
  output CARRYCASCOUT,
  ...

  input [29:0] A,
  input [29:0] ACIN,
  input [3:0] ALUMODE,
  input [17:0] B,
  input [17:0] BCIN,
  input [47:0] C,
  ...
);
```

Configuring the DSP requires setting 100+ ports and parameters

Table 2-4: **OPMODE Control Bits Select X Multiplexer Outputs**

W OPMODE[8:7]	Z OPMODE[6:4]	Y OPMODE[3:2]	X OPMODE[1:0]	X Multiplexer Output	Notes
xx	xxx	xx	00	0	Default
xx	xxx	01	01	M	Must select with OPMODE[3:2] = 01
xx	xxx	xx	10	P	Requires PREG = 1
xx	xxx	xx	11	A:B	48-bits wide

When either TWO24 or FOUR12 mode is selected, the multiplier must not be used, and USE_MULT must be set to NONE.

Notes:

1. When these data pins are not used and to reduce leakage power dissipation, the data pin input signals must be tied High, the input register must be selected, and the CE and RST input control signals must be tied Low. An example of unused C input recommended settings would be setting C[47:0] = all ones, CREG = 1, CEC = 0, and RSTC = 0.
2. These signals are dedicated routing paths internal to the DSP48E2 column. They are not accessible via general routing resources.
3. All signals are active High.

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  parameter integer ALUMODEREG = 1,
  parameter AMULTSEL = "A",
  parameter integer AREG = 1,
  parameter AUTORESET_PATDET = "NO_RESET",
  parameter AUTORESET_PRIORITY = "RESET",
  parameter A_INPUT = "DIRECT",
  ...
)(
  output [29:0] ACOUT,
  output [17:0] BCOUT,
  output CARRYCASCOUT,
  ...

  input [29:0] A,
  input [29:0] ACIN,
  input [3:0] ALUMODE,
  input [17:0] B,
  input [17:0] BCIN,
  input [47:0] C,
  ...
);
```

Configuring the DSP requires setting 100+ ports and parameters

Table 2-4: **OPMODE Control Bits Select X Multiplexer Outputs**

W OPMODE[8:7]	Z OPMODE[6:4]	Y OPMODE[3:2]	X OPMODE[1:0]	X Multiplexer Output	Notes
xx	xxx	xx	00	0	Default
xx	xxx	01	01	M	Must select with OPMODE[3:2] = 01
xx	xxx	xx	10	P	Requires PREG = 1
xx	xxx	xx	11	A:B	48-bits wide

When either TWO24 or FOUR12 mode is selected, the multiplier must not be used, and USE_MULT must be set to NONE.

Notes:

1. When these data pins are not used and to reduce leakage power dissipation, the data pin input signals must be tied High, the input register must be selected, and the CE and RST input control signals must be tied Low. An example of unused C input recommended settings would be setting C[47:0] = all ones, CREG = 1, CEC = 0, and RSTC = 0.
2. These signals are dedicated routing paths internal to the DSP48E2 column. They are not accessible via general routing resources.
3. All signals are active High.

Configuring a DSP sounds a lot like writing a program!

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  par
  par
  par
  par
  par
  ...
)(
  out
  out
  out
  ...

  inp
  inp
  inp
  inp
  inp
  input [47:0] C,
  ...
);
```

Table 2-4: **OPMODE Control Bits Select X Multiplexer Outputs**

W OPMODE[8:7]	Z OPMODE[6:4]	Y OPMODE[3:2]	X OPMODE[1:0]	X Multiplexer Output	Notes

Insight #1: configuring DSPs and other complex primitives is similar to writing a program…

Our first insight is that…

[build] so why not use program synthesis?

For those unfamiliar with program synthesis, I won't go into too much detail, but just know that [build] *solver aided program synthesis* is the process of…

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  par
  par
  par
  par
  par
  ...
)(
  out
  out
  out
  ...

  inp
  inp
  inp
  inp
  inp
  input [47:0] C,
  ...
);
```

Table 2-4: **OPMODE Control Bits Select X Multiplexer Outputs**

W OPMODE[8:7]	Z OPMODE[6:4]	Y OPMODE[3:2]	X OPMODE[1:0]	X Multiplexer Output	Notes

Insight #1: configuring DSPs and other complex primitives is similar to writing a program…

…so use *program synthesis.*

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  par
  par
  par
  par
  par
  ...
)(
  out
  out
  out
  ...

  inp
  inp
  inp
  inp
  inp
  input [47:0] C,
  ...
);
```

Table 2-4: **OPMODE Control Bits Select X Multiplexer Outputs**

W OPMODE[8:7]	Z OPMODE[6:4]	Y OPMODE[3:2]	X OPMODE[1:0]	X Multiplexer Output	Notes

Insight #1: configuring DSPs and other complex primitives is similar to writing a program…

…so use *program synthesis.*

Solver-aided program synthesis: using SMT/SAT/etc. to generate programs by solving a set of constraints.

```
module DSP48E2 #(
  parameter integer ACASCREG = 1,
  parameter integer ADREG = 1,
  parameter integer ALUMODEREG = 1,
  parameter AMULTSEL = "A",
  parameter integer AREG = 1,
  parameter AUTORESET_PATDET = "NO_RESET",
  parameter AUTORESET_PRIORITY = "RESET",
  parameter A_INPUT = "DIRECT",
  ...
)(
  output [29:0] ACOUT,
  output [17:0] BCOUT,
  output CARRYCASCOUT,
  ...

  input [29:0] A,
  input [29:0] ACIN,
  input [3:0] ALUMODE,
  input [17:0] B,
  input [17:0] BCIN,
  input [47:0] C,
  ...
);
```

Table 2-4: **OPMODE Control Bits Select X Multiplexer Outputs**

W OPMODE[8:7]	Z OPMODE[6:4]	Y OPMODE[3:2]	X OPMODE[1:0]	X Multiplexer Output	Notes
xx	xxx	xx	00	0	Default
xx	xxx	01	01	M	Must select with OPMODE[3:2] = 01
xx	xxx	xx	10	P	Requires PREG = 1
xx	xxx	xx	11	A:B	48-bits wide

When either TWO24 or FOUR12 mode is selected, the multiplier must not be used, and USE_MULT must be set to NONE.

Notes:

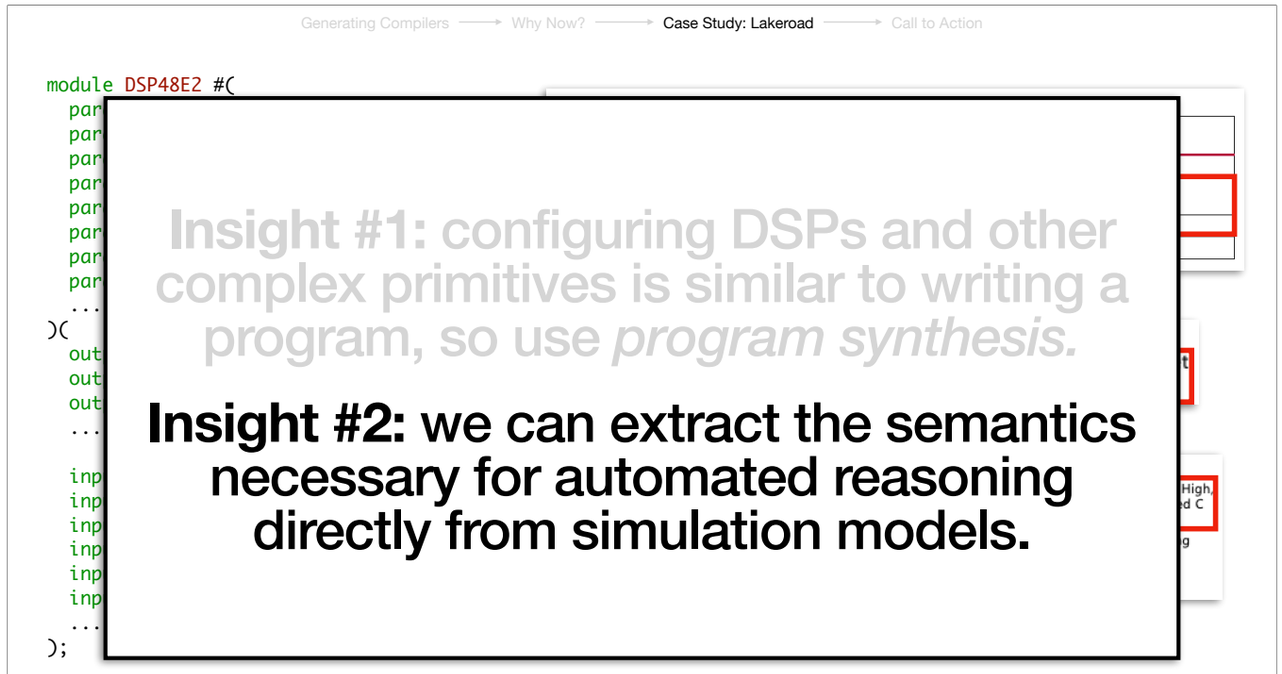
1. When these data pins are not used and to reduce leakage power dissipation, the data pin input signals must be tied High, the input register must be selected, and the CE and RST input control signals must be tied Low. An example of unused C input recommended settings would be setting C[47:0] = all ones, CREG = 1, CEC = 0, and RSTC = 0.
2. These signals are dedicated routing paths internal to the DSP48E2 column. They are not accessible via general routing resources.
3. All signals are active High.

Because configuring DSPs is so complicated, Xilinx provides…

a 1500 line simulation model of the DSP, which hardware designers use to validate their designs.
But this is useful to us as well, because
[build] …

DSP48E2.v

Simulation models provide the formal semantics of behaviors and constraints necessary for automated reasoning!



This leads us to our second insight, which is that we can …

Lakeroad: a hardware synthesis tool utilizing program synthesis and semantics extracted from simulation models to target complex, programmable FPGA primitives.

Using these two insights, we build Lakeroad, which is…

Workload	Signed?	# Stages	Yosys	SOTA	Lakeroad
`((d + a) * b) | c`	X	1	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP
`((d - a) * b) | c`	✓	2	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP
`((d - a) * b) ^ c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP
`((d + a) * b) & c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP
`((d + a) * b) ^ c`	X	2	1 DSP, 18 LUT	1 DSP, 9 LUT	1 DSP

Initial results with Lakeroad suggest that Lakeroad is able to find mappings that other tools do not, successfully mapping designs onto
[build] a single DSP when other tools
[build] fail to do so.

Workload	Signed?	# Stages	Yosys	SOTA	Lakeroad
`((d + a) * b) | c`	X	1	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP
`((d - a) * b) | c`	✓	2	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP
`((d - a) * b) ^ c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP
`((d + a) * b) & c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP
`((d + a) * b) ^ c`	X	2	1 DSP, 18 LUT	1 DSP, 9 LUT	1 DSP

Workload	Signed?	# Stages	Yosys	SOTA	Lakeroad
`((d + a) * b) | c`	X	1	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP
`((d - a) * b) | c`	✓	2	1 DSP, 20 LUT	1 DSP, 10 LUT	1 DSP
`((d - a) * b) ^ c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP
`((d + a) * b) & c`	✓	3	1 DSP, 22 LUT	2 DSP, 11 LUT	1 DSP
`((d + a) * b) ^ c`	X	2	1 DSP, 18 LUT	1 DSP, 9 LUT	1 DSP

How does Lakeroad support the thesis of this talk?
We claim that
[build] …
Lakeroad exemplifies this directly, by
[build] automatically enabling compilation…

Furthermore, we claim that
[build] …
Which Lakeroad demonstrates by showing that
[build] automated methods are…

Compilers should be generated from formal models of hardware.

Compilers should be generated from formal models of hardware.

Lakeroad automatically enables compilation to FPGA primitives, given the simulation models of those primitives.

Compilers should be generated from formal models of hardware.

Lakeroad automatically enables compilation to FPGA primitives, given the simulation models of those primitives.

With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

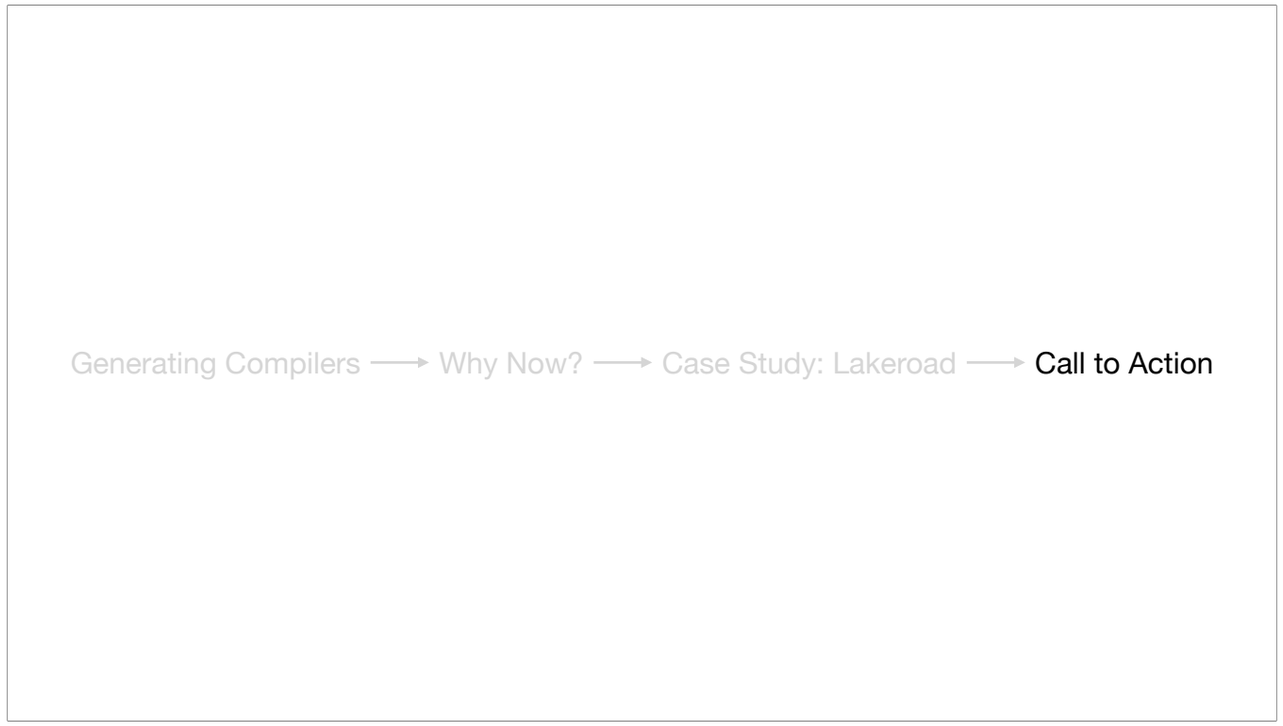
Compilers should be generated from formal models of hardware.

Lakeroad automatically enables compilation to FPGA primitives, given the simulation models of those primitives.

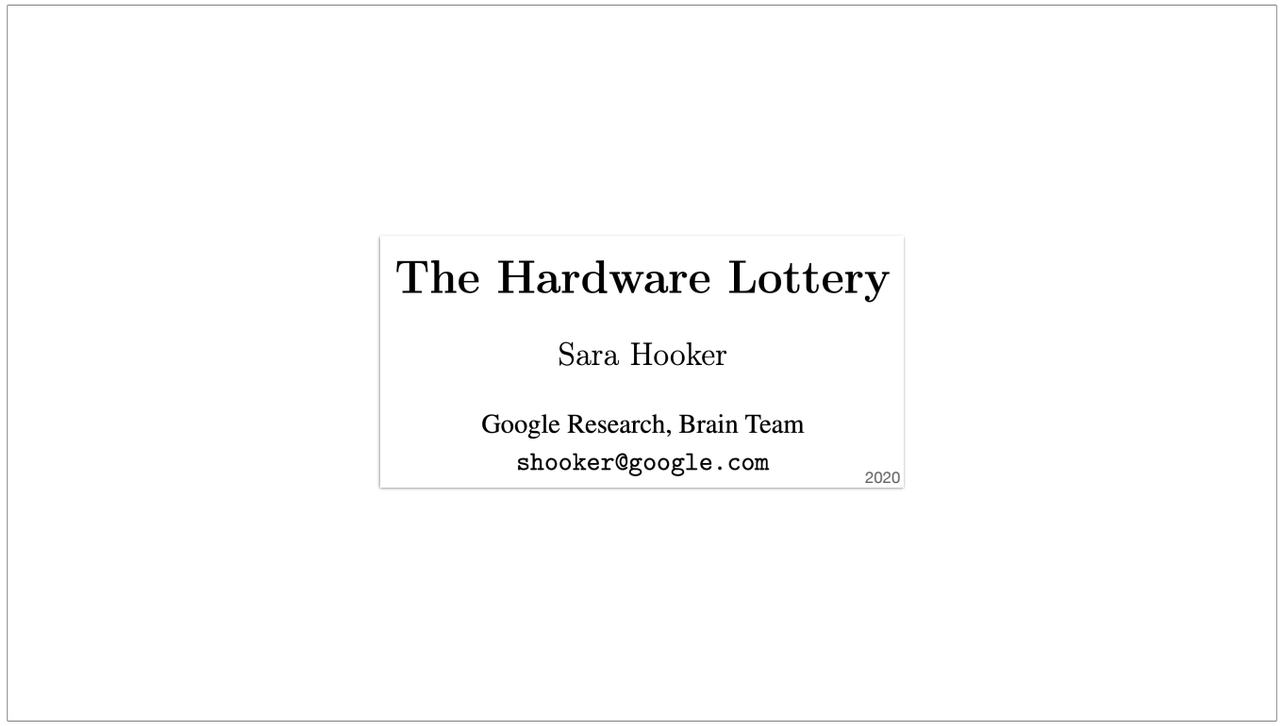
With the growing diversity of hardware and the rapid improvement of automated reasoning, now is the time to make this a reality.

Lakeroad demonstrates that automated methods are now powerful enough address gaps in existing state-of-the-art tools.

Now finally,



Let’s conclude with a call to action.



In conclusion, I want to end where we began.
I believe the hardware lottery is a direct challenge to our community.

If there's one message I want to leave you with today, it's this:
[build] It's on on all of us in this room to fight against the hardware lottery, by making sure that practitioners in all fields have the hardware and compilers they need to advance their research.

What I've proposed today—automatically generating compilers from formal models of hardware—is just one possible solution. Whether or not you agree with the solution, I hope you'll agree with the larger goal of ending the hardware lottery once and for all.

The Hardware Lottery

Sara Hooker

Google Research, Brain Team
shooker@google.com

2020

It's on on all of us to fight against the hardware lottery, by making sure that practitioners in all fields have the hardware and compilers they need to advance their research.

Thank you!

Thank you, everybody!