KEVIN LOUGHLIN

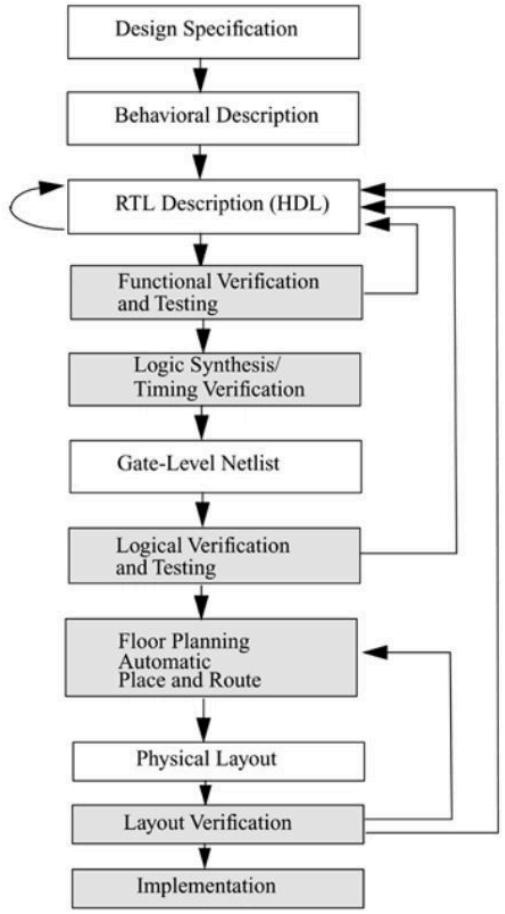
IAN NEAL

REUSABILITY IS FIRRTL GROUND:

HARDWARE CONSTRUCTION LANGUAGES, COMPILER FRAMEWORKS, AND TRANSFORMATIONS

SUMMARY: WHAT ARE WE TALKING ABOUT?

- Designing chip layouts for ASICs and FPGAs is hard
  - Current programming methods have low flexibility and promote little code reuse
- We want to bring hardware design to where software design currently is
  - Lots of flexibility in programming with high-level languages
  - Lots of code reuse through common libraries
  - An easier way to write generic code that can be retargeted to different hardware platforms



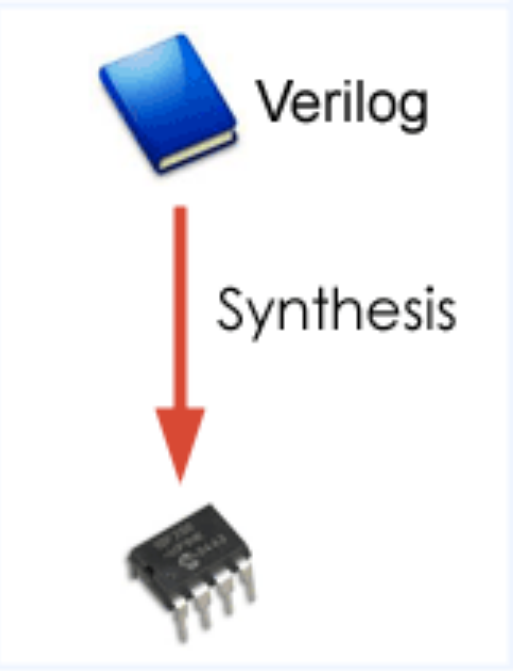
Source: asic4u.wordpress.com

WHY ARE WE WRITING CODE FOR *HARDWARE*?

- End of Moore's Law and Dennard Scaling
  - Can't just make transistors smaller and increase clock speed
  - Need to make better use of current transistor densities
- ASICs & FPGAs
  - Tailor hardware to efficiently solve a specific problem
- Laying out gate by hand is not scalable for modern chips
  - Like how no one wants to write assembly in binary!

LET'S TALK (TRADITIONAL) HARDWARE COMPILATION!

- Not compilation, but *synthesis*
  - Rather than C => bitcode, turn Verilog => layout of gates
  - I.e., create a physical instantiation of the circuit your code describes
- Synthesis can be trickier than software compilation
  - Many physical restraints (space, number of resources, etc.)
  - If your abstract circuit is not physically possible, synthesis fails



Source: doulos.com

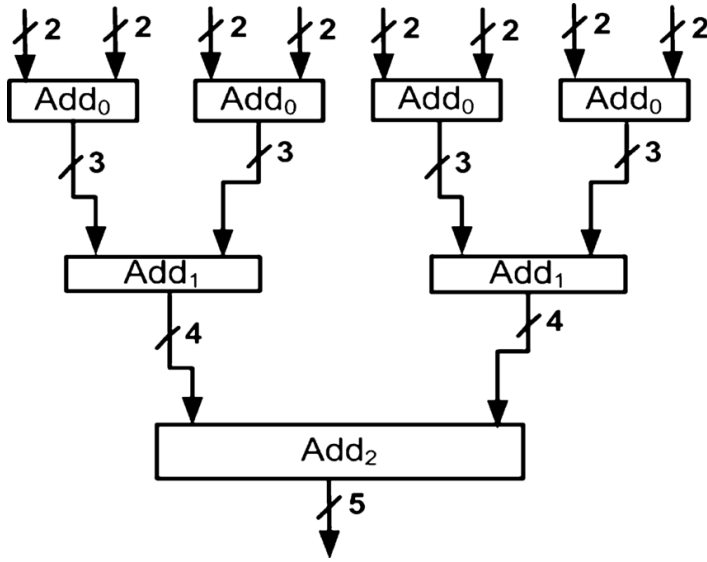
LET'S TALK (TRADITIONAL) HARDWARE DESIGN!

- Circuits described at Register Transfer Level (RTL)
  - Register = an object that can hold a value (state)
  - RTL models circuits in terms of the signals flowing between registers, and the operations performed on those signals
- Hardware Design Languages (HDLs) express circuits at RTL level
  - E.g., Verilog or VHDL
- Scientifically-speaking, these languages are "meh"

```
1   module up_counter    (
2   out     ,  // Output of the counter
3   enable  ,  // enable for counter
4   clk     ,  // clock Input
5   reset      // reset Input
6   );
7   //----------Output Ports--------------
8       output [7:0] out;
9   //------------Input Ports--------------
10      input enable, clk, reset;
11  //------------Internal Variables---------
12      reg [7:0] out;
13  //-------------Code Starts Here-------
14  always @(posedge clk)
15  if (reset) begin
16    out <= 8'b0 ;
17  end else if (enable) begin
18    out <= out + 1;
19  end
20  endmodule
```

WHAT'S WRONG WITH HDLS?

- HDLs don't (easily) allow for good software engineering practices
  - Code re-use
  - Parametrization
  - Readability



Source: Tayab Memon

- Example: adder-reduction tree (pictured)
  - Verilog and VHDL can't express recursive generate statements
  - Designer must manually unroll loop and calculate indices for all instances
- Different platforms require different HDL code
  - Example: FPGAs and ASICs must use different memory interfaces

GOALS

- Want to enable high levels of code reuse like the software community has
  - Reuse hardware libraries across projects
  - Reuse hardware libraries or projects across platforms
- Want to enable easy development for new hardware projects
  - These software people can go from idea to implementation in under two weeks! *(The hardware people are jealous)*

SOLUTION PART 1: HCLS!

Source: twitter.com

- HCLs = Hardware ***Construction*** Languages
  - Instead of writing out every detail of the circuit, write a program to generate the circuit you want
  - Essentially a way to autogenerate HDL code.
- These already exist (Chisel) and are used to make up for features that HDLs lack (parameterization, recursion, etc.)

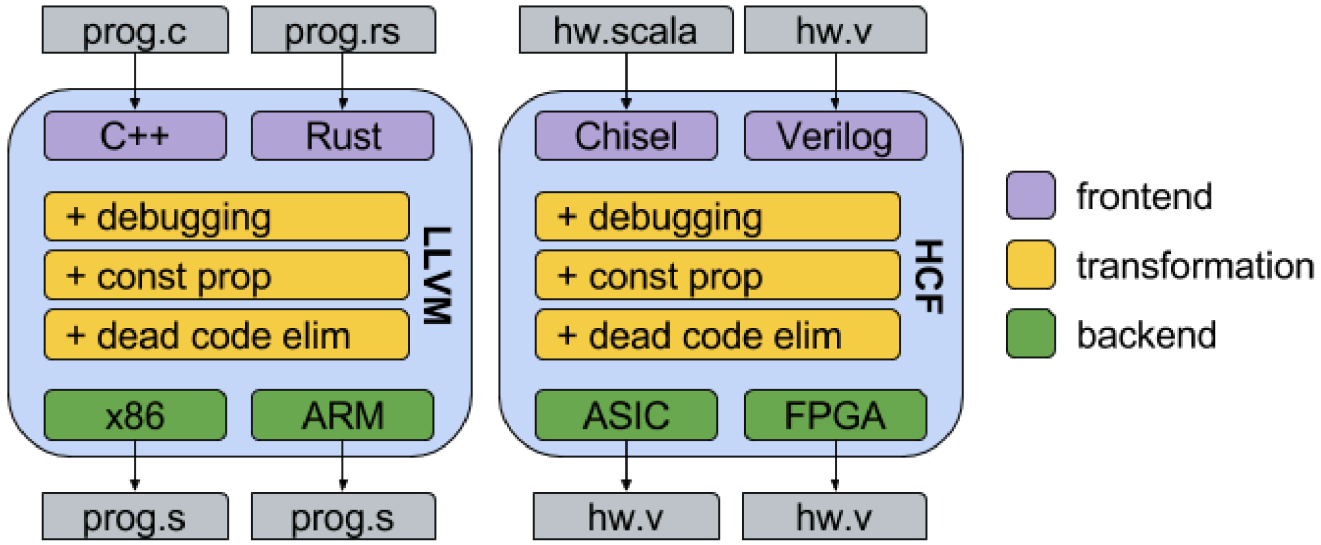
SOLUTION
PART 2: HCFS!

Source: github.com

- HCF = Hardware ***Compiler Framework***
  - Like LLVM, but for RTL
- Enables us to have a compiler *backend* which takes generic RTL and emit platform-specific Verilog
- One key difference between HCFs and LLVM: HCFs emit Verilog code which is given to proprietary compilers, LLVM emits binaries which are ready to run.

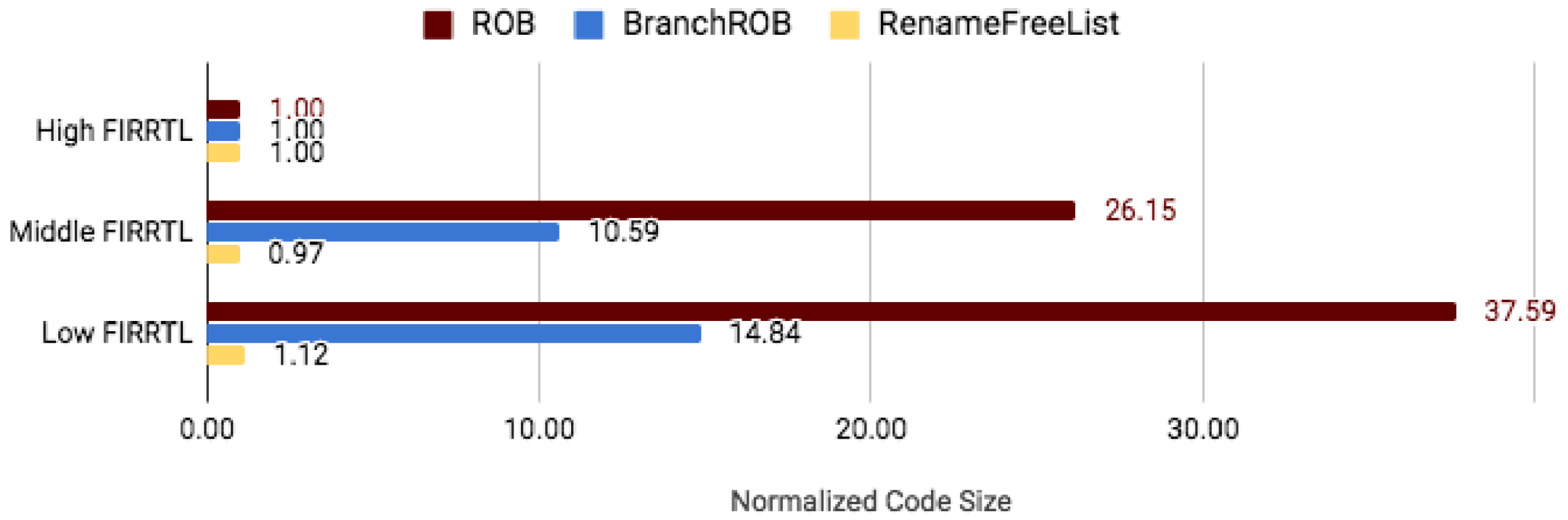
INTRODUCING: FIRRTL

- Flexible Intermediate Representation for RTL (FIRRTL)
- Allows for shared transformations
- Platform-specific backends allow for generic libraries to be use
- Emits Verilog that can be consumed by downstream compilers and optimizers

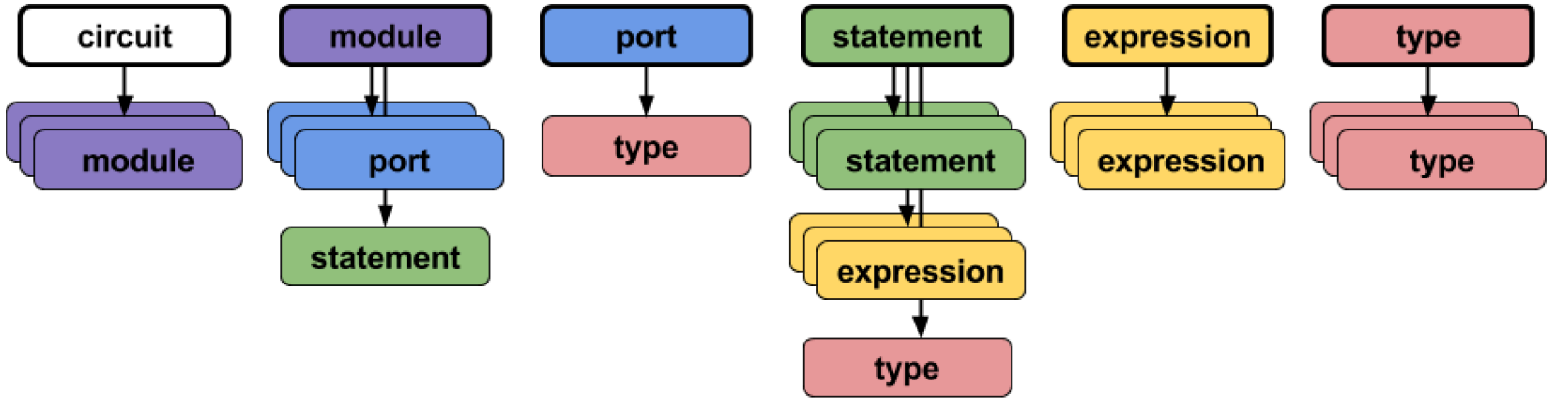


INTERNAL REPRESENTATION

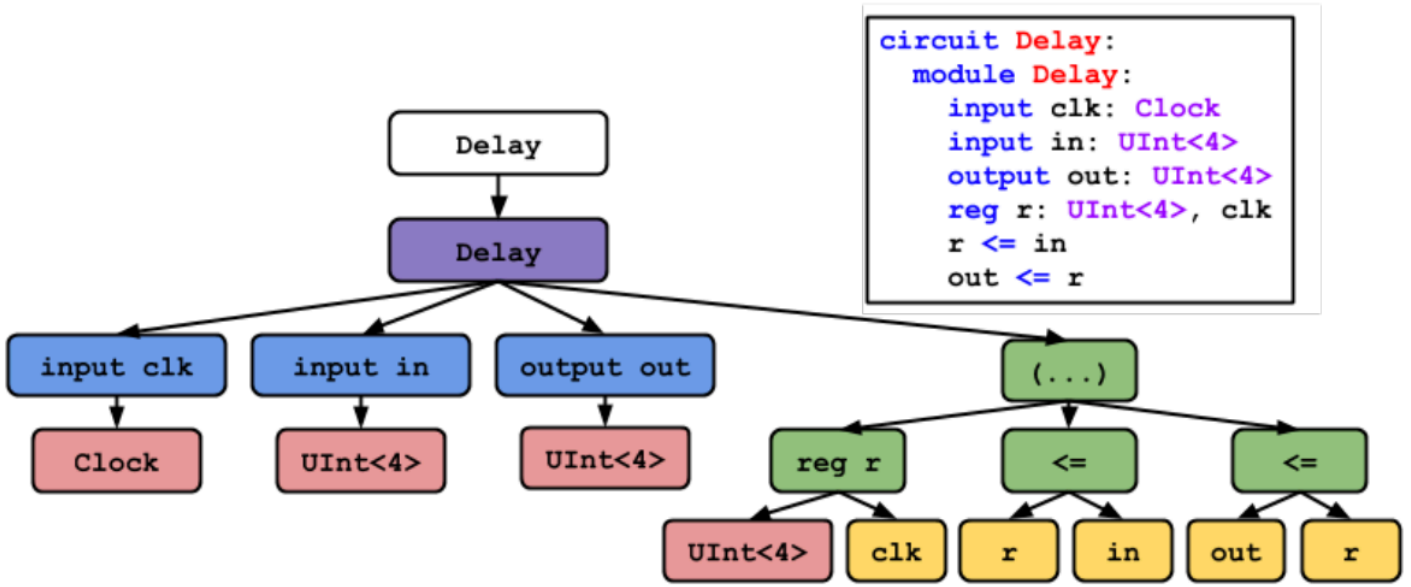
- Has three internal forms of representation
  - High, middle, low, ranging from feature rich to poor
  - Low form maps directly to Verilog
  - Optimizations specify which form they use, can later be elevated or lowered
- Operates on an Abstract Syntax Tree (AST) of different hardware types
  - *circuit, module, port, statement, expression, type*



COMPARING HIGH, MIDDLE, AND LOW FORMS OF FIRRTL


circuit
module
module
port
statement
port
type
statement
statement
expression
type
expression
expression
type
type

FIRRTL AST


circuit Delay:
module Delay:
input clk: Clock
input in: UInt<4>
output out: UInt<4>
reg r: UInt<4>, clk
r <= in
out <= r
Delay
Delay
input clk
input in
output out
(...)
Clock
UInt<4>
UInt<4>
reg r
<=
<=
UInt<4>
clk
r
in
out
r

APPLYING TRANSFORMATIONS

- Each IR node has a *map* function
  - Applies optimizations to itself and all children that match the input-output specification
- Example: simplification transformations
  - Replaces the bulk connector (connects aggregate types) with connections to individual primitives (as Verilog doesn't support the bulk connector)
- Most transformations handle simplification, readability, or analysis/instrumentation
  - Downstream compilers handle aggressive logic optimizations

EVALUATION

- Many case studies
- Provides a generic optimization to increase utilization of block RAM
  - ***Reduces BRAM utilization by 3x***
- By using FIRRTL, made switching ASIC from register design to SRAM design trivial
  - ***Saves up to 6x area***
- Wrote a parameterized version of RocketChip
  - ***94% of the design was reused***

QUESTIONS?