Design of the SUS Language

Lennart Van Hirtum
PC2 - Paderborn University

Prof. Christian Plessl
PC2 - Paderborn University

Outline

- Motivation & where others fall short
- SUS Design Process
- Example SUS Code
- Typing
- Latency Counting
- Implementation & Future
- Live Demo

Motivation

What I want out of an HDL

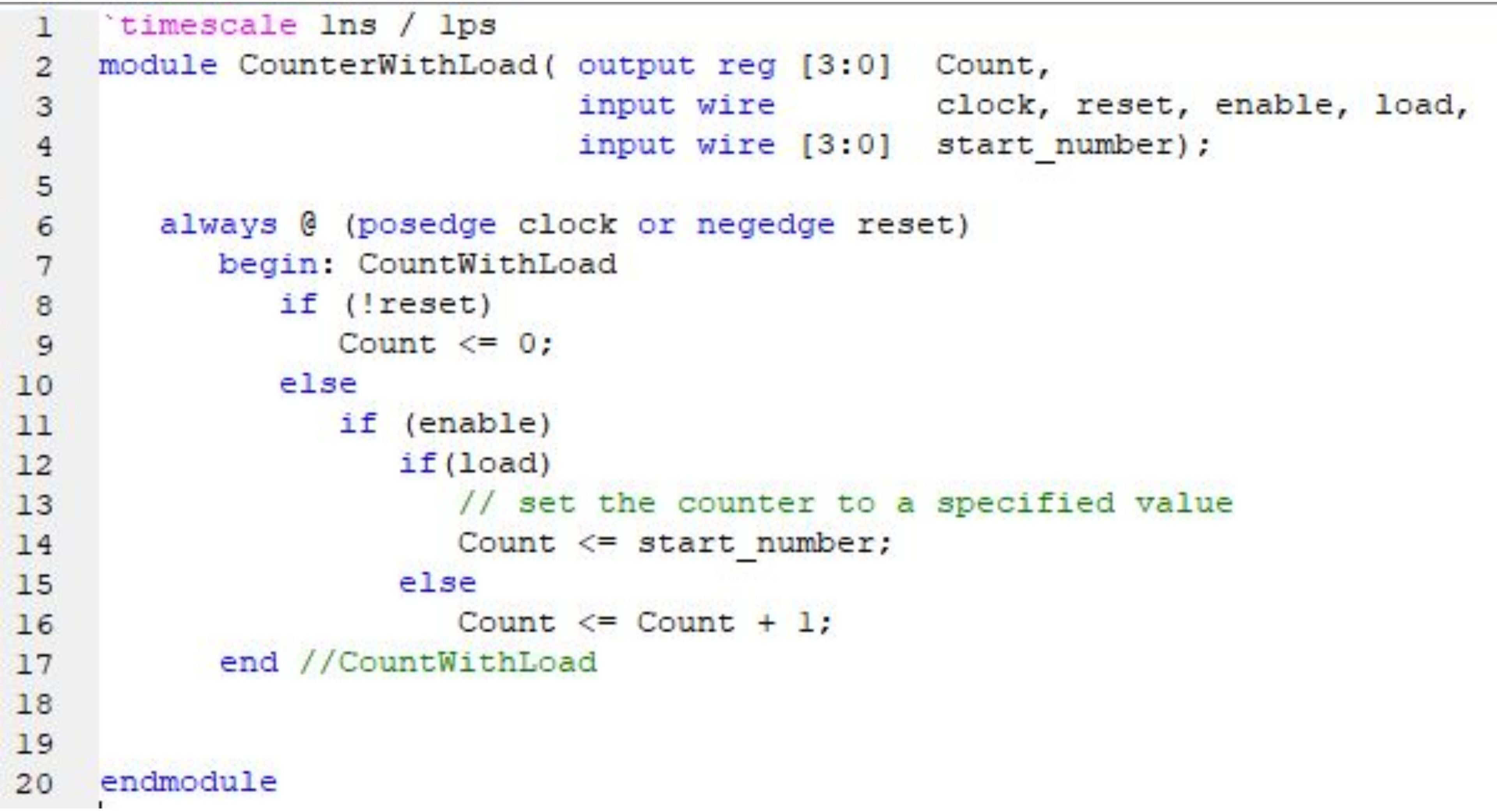
- Full control over generated hardware
- All HDL code must be Synthesizeable
  - Keep your simulations in software like CocoTB!
- Language must have notion of Cycle-wise timing
- First-class Pipelining support
- First-class IDE support

Where do other languages fall short?

- **(System)Verilog, VHDL?**

Where do other languages fall short?

- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware



```verilog
`timescale 1ns / 1ps
module CounterWithLoad( output reg [3:0]  Count,
                        input wire        clock, reset, enable, load,
                        input wire [3:0]  start_number);

   always @ (posedge clock or negedge reset)
      begin: CountWithLoad
         if (!reset)
            Count <= 0;
         else
            if (enable)
               if(load)
                  // set the counter to a specified value
                  Count <= start_number;
               else
                  Count <= Count + 1;
      end //CountWithLoad


endmodule
```

Where do other languages fall short?

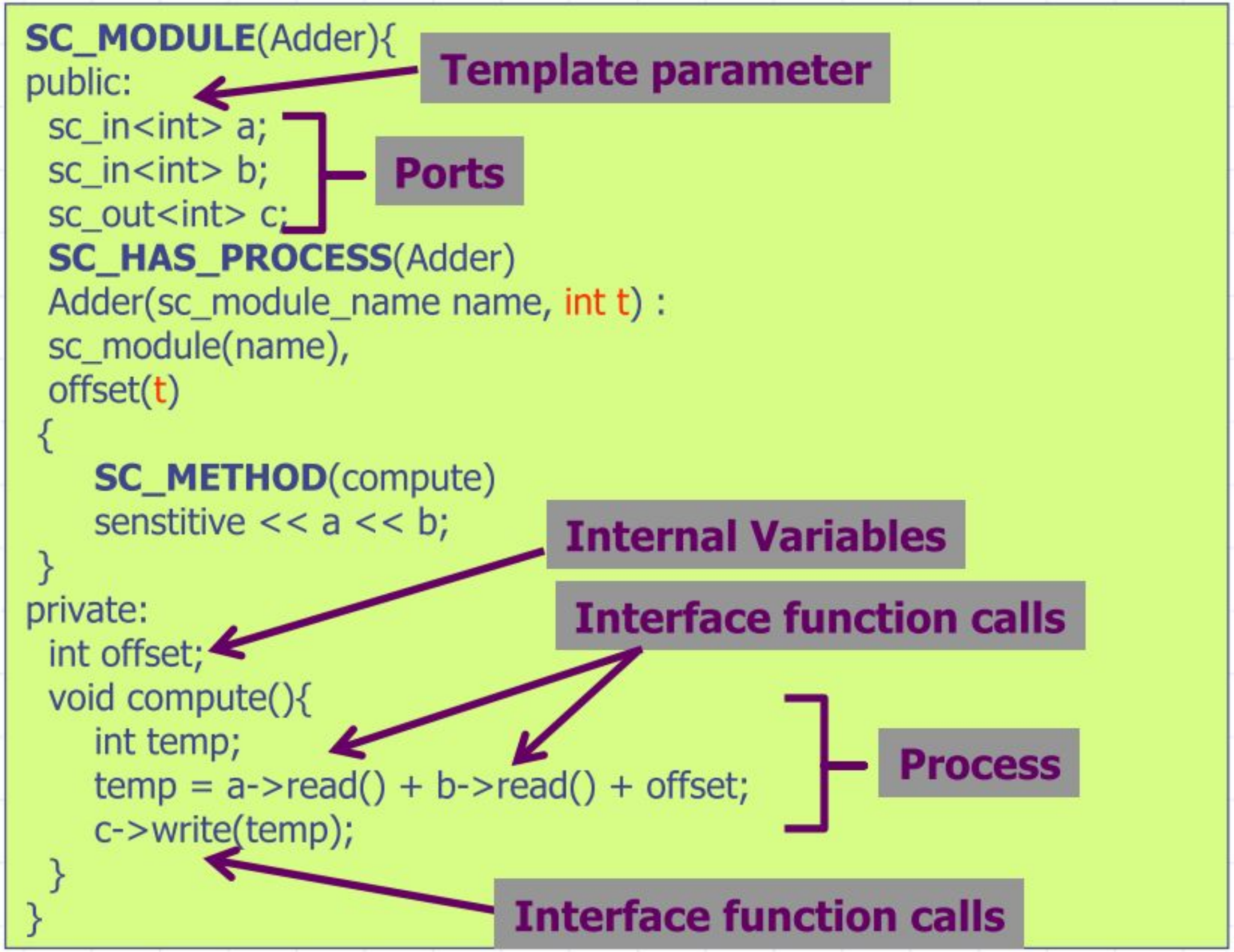
- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware

Where do other languages fall short?

- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**

Where do other languages fall short?

- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support



```
SC_MODULE(Adder){
public:
  sc_in<int> a;
  sc_in<int> b;
  sc_out<int> c;
  SC_HAS_PROCESS(Adder)
  Adder(sc_module_name name, int t) :
  sc_module(name),
  offset(t)
 {
     SC_METHOD(compute)
     senstitive << a << b;
 }
private:
  int offset;
  void compute(){
     int temp;
     temp = a->read() + b->read() + offset;
     c->write(temp);
  }
}
```

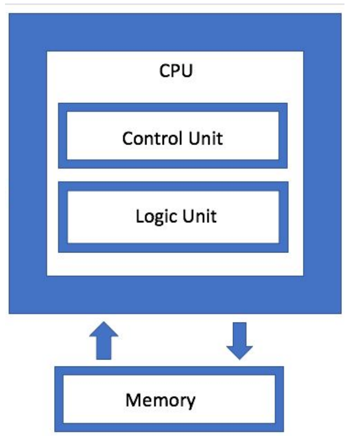
```
class MemFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T, depth: Int) {

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth-1).U, 0.U, cntReg + 1.U)
    when (incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  val mem = SyncReadMem(depth, gen)

  val incrRead = WireInit(false.B)
  val incrWrite = WireInit(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)
```

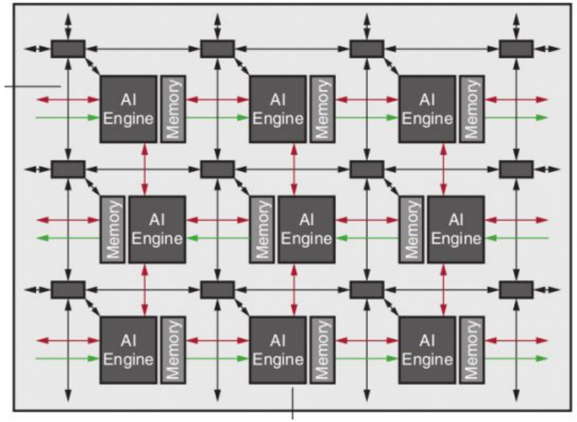
Where do other languages fall short?

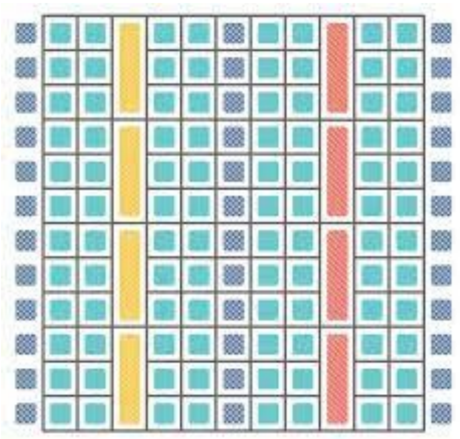
- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support

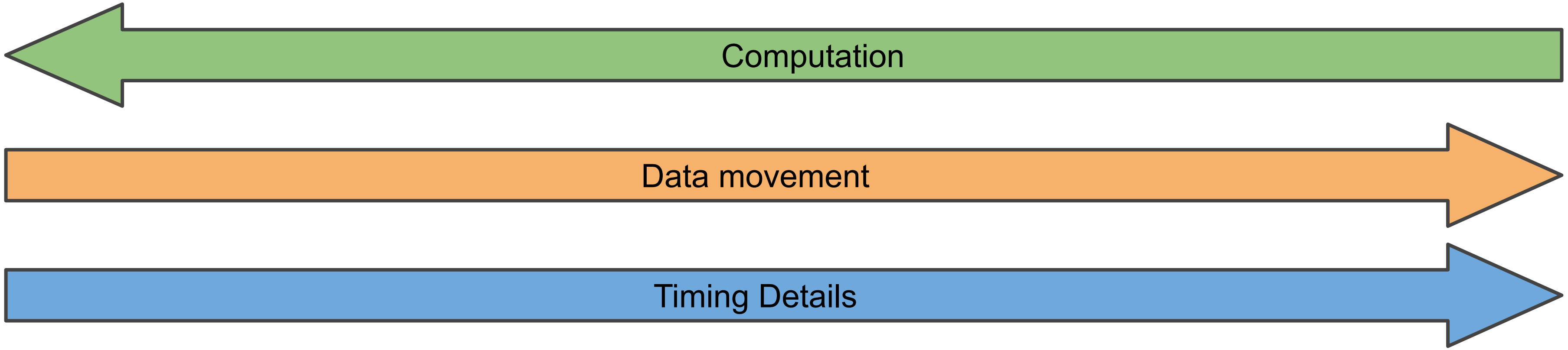
Where do other languages fall short?

- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support
- **HLS?**

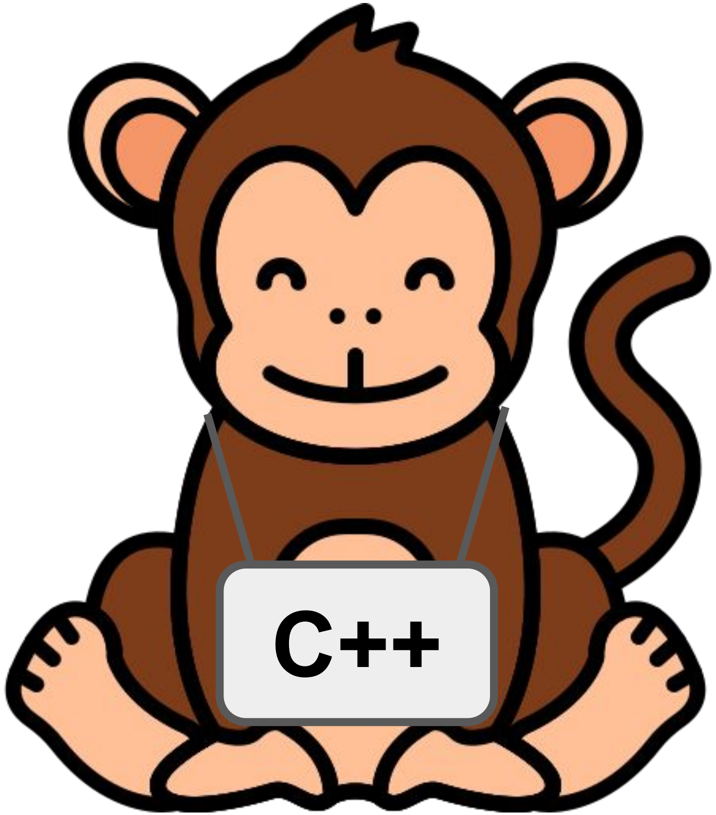


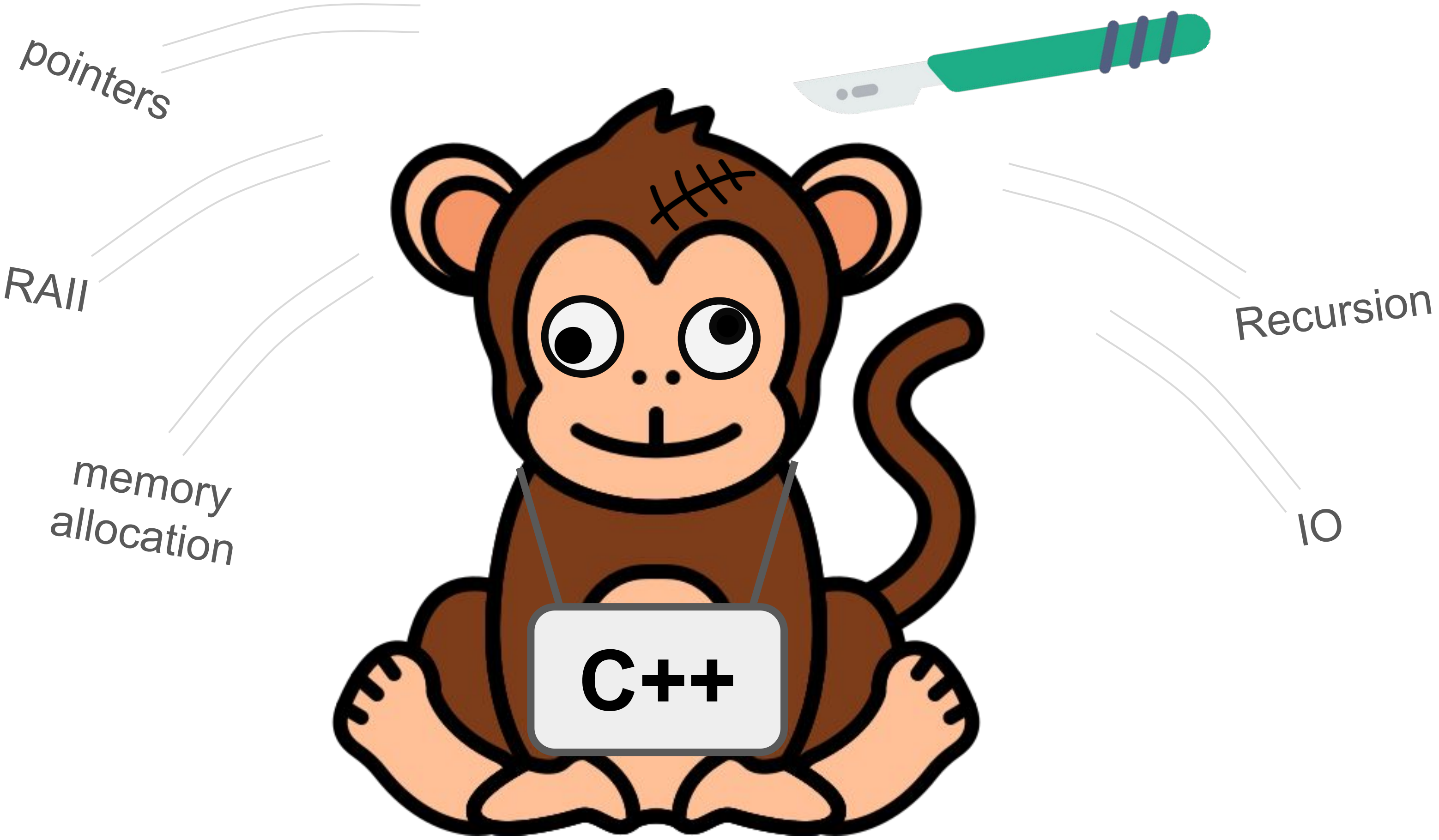






HLS: We can do it all!


C++

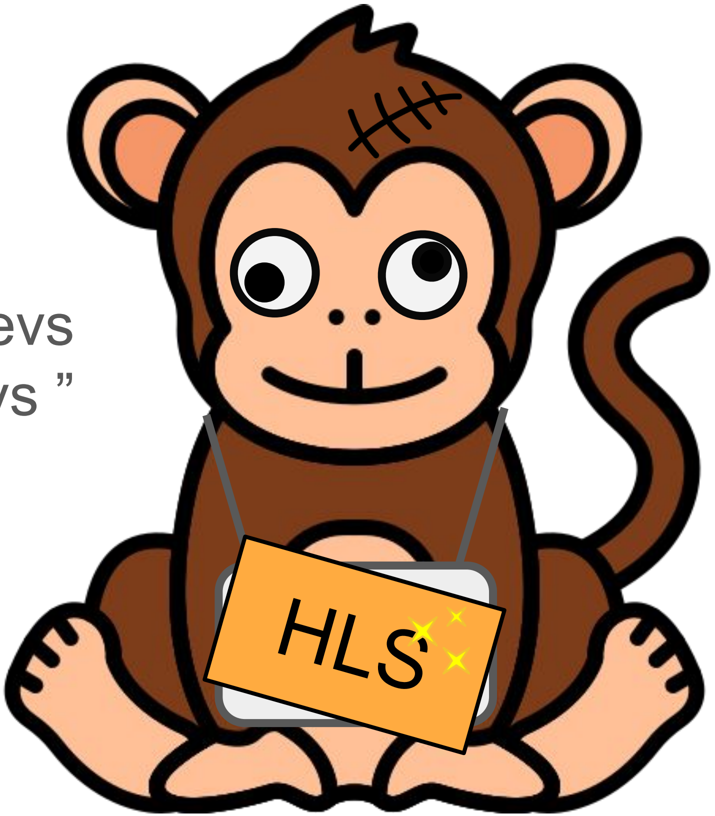

pointers
RAII
memory allocation
Recursion
IO
C++

“ Single-source host and kernel ”

“ Turn software devs into hardware devs ”

“ It improves portability ”

“ The compiler will figure it out ”



Where do other languages fall short?

- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support
- **HLS?**

Where do other languages fall short?

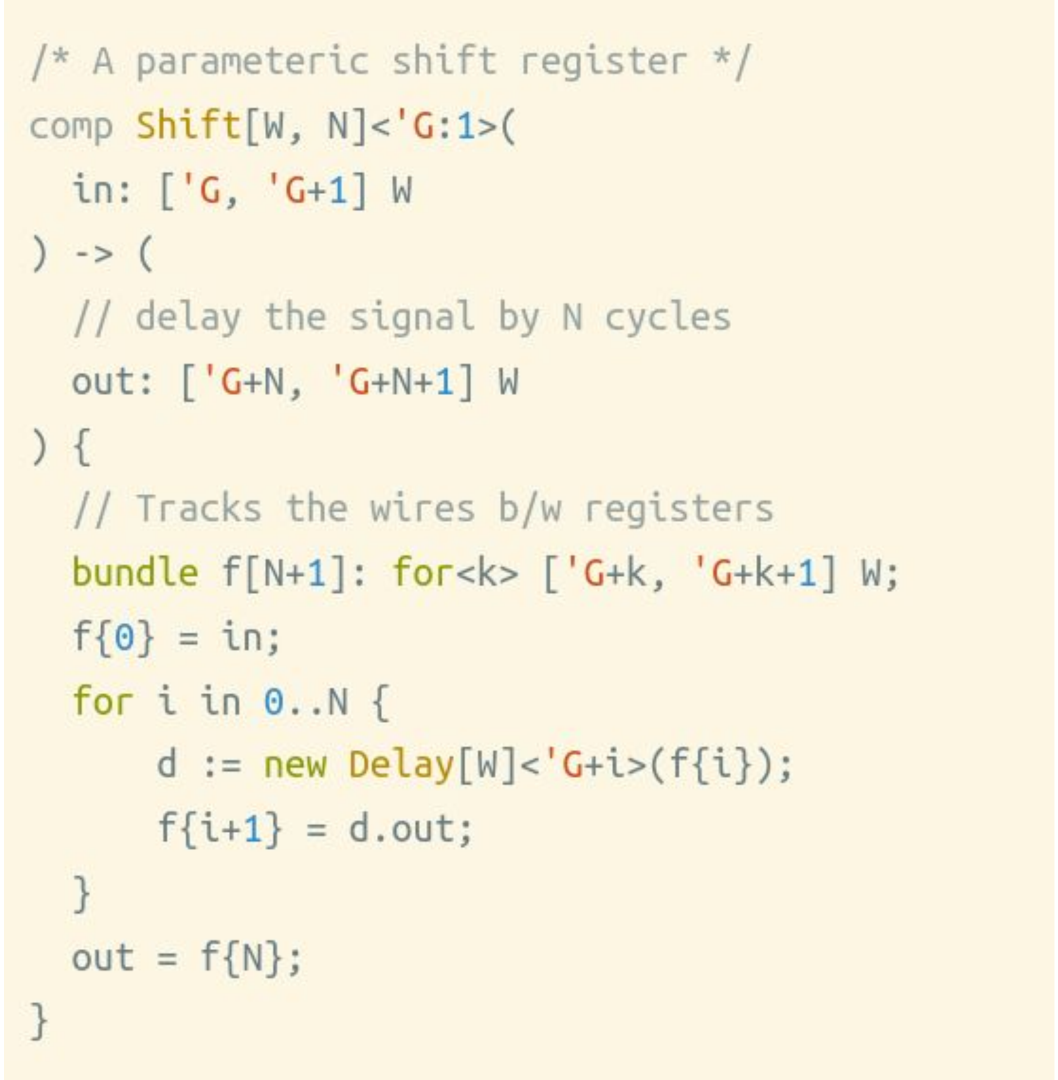
- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support
- **HLS?**
  - Lose control over generated hardware (Are you sure it's optimal?)
  - (Corporate HLS) is not portable at all, and stifles lower-level access for competitors

Where do other languages fall short?

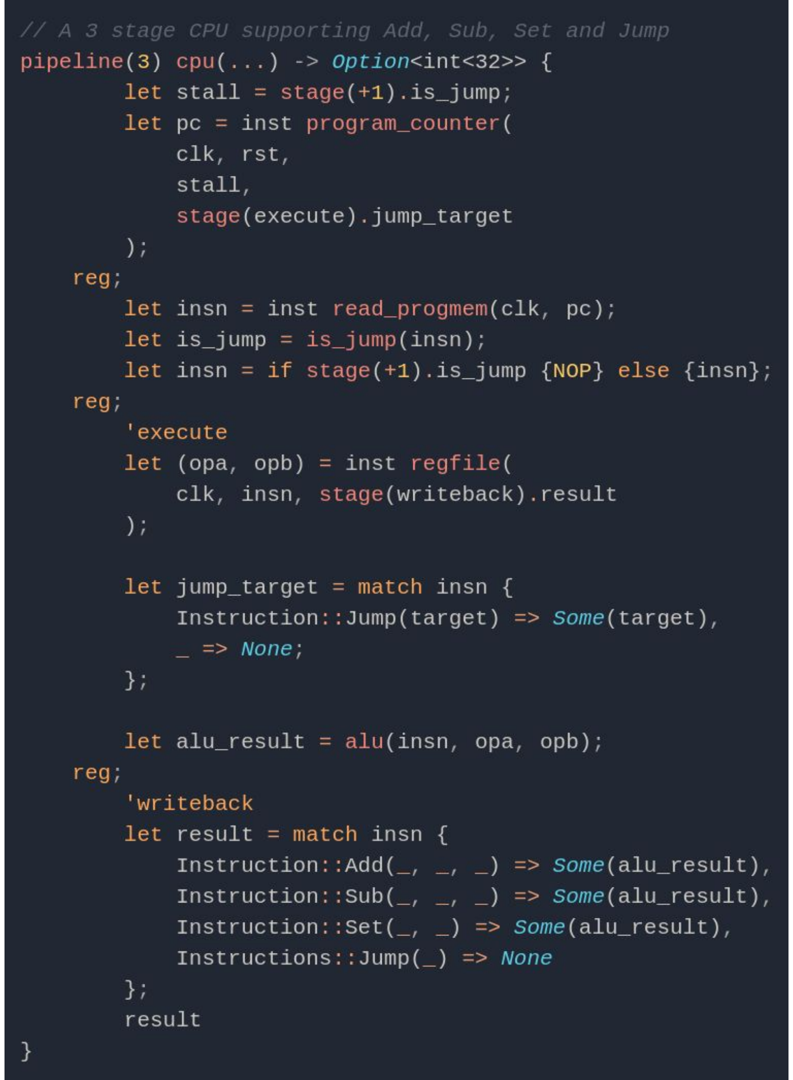
- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support
- **HLS?**
  - Lose control over generated hardware (Are you sure it's optimal?)
  - (Corporate HLS) is not portable at all, and stifles lower-level access for competitors
- **Other NeoHDLs (Filament & Spade)?**

Where do other languages fall short?

- **(System)Verilog, VHDL?**
  - *Very* verbose
  - No defence against incorrect hardware
- **Embedded Generation Languages (Chisel in Scala, SystemC in C++)?**
  - Suffer under parent language Syntax
  - Little defence against incorrect hardware
  - No hardware-specific IDE support
- **HLS?**
  - Lose control over generated hardware (Are you sure it’s optimal?)
  - (Corporate HLS) is not portable at all, and stifles lower-level access for competitors
- **Other NeoHDLs (Filament & Spade)?**
  - Focus on Correctness
  - Focus on Timing



```
/* A parameteric shift register */
comp Shift[W, N]<'G:1>(
  in: ['G, 'G+1] W
) -> (
  // delay the signal by N cycles
  out: ['G+N, 'G+N+1] W
) {
  // Tracks the wires b/w registers
  bundle f[N+1]: for<k> ['G+k, 'G+k+1] W;
  f{0} = in;
  for i in 0..N {
      d := new Delay[W]<'G+i>(f{i});
      f{i+1} = d.out;
  }
  out = f{N};
}
```



```
// A 3 stage CPU supporting Add, Sub, Set and Jump
pipeline(3) cpu(...) -> Option<int<32>> {
        let stall = stage(+1).is_jump;
        let pc = inst program_counter(
            clk, rst,
            stall,
            stage(execute).jump_target
        );
    reg;
        let insn = inst read_progmem(clk, pc);
        let is_jump = is_jump(insn);
        let insn = if stage(+1).is_jump {NOP} else {insn};
    reg;
        'execute
        let (opa, opb) = inst regfile(
            clk, insn, stage(writeback).result
        );

        let jump_target = match insn {
            Instruction::Jump(target) => Some(target),
            _ => None;
        };

        let alu_result = alu(insn, opa, opb);
    reg;
        'writeback
        let result = match insn {
            Instruction::Add(_, _, _) => Some(alu_result),
            Instruction::Sub(_, _, _) => Some(alu_result),
            Instruction::Set(_, _) => Some(alu_result),
            Instructions::Jump(_) => None
        };
        result
}
```

Hot Takes

Hot Takes

There are no reusable hardware components

Software prototyping for hardware doesn’t exist
(unless it’s named Verilator)

HLS is not the solution

So… Abstraction Bad?

Of course not.

Abstraction Good!

Good Abstractions

Represent the programmer’s intent

Replace error-prone constructs

Don’t get in the way

⇒ Reduce Designer mental load

Bad Abstractions

Trade in design freedom for “abstraction”

⇒ Contort the Designer into the vendor’s paradigm

SUS Design Process

Control

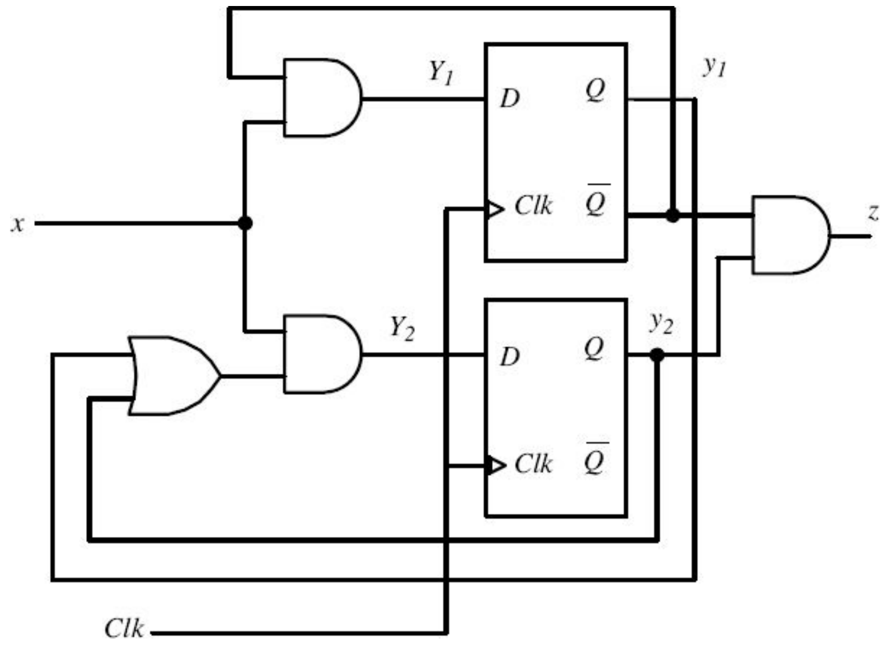
Feedback

Pragmatism

Control

If I can draw it, I want to be able to write it

⇒ All synchronous hardware representable



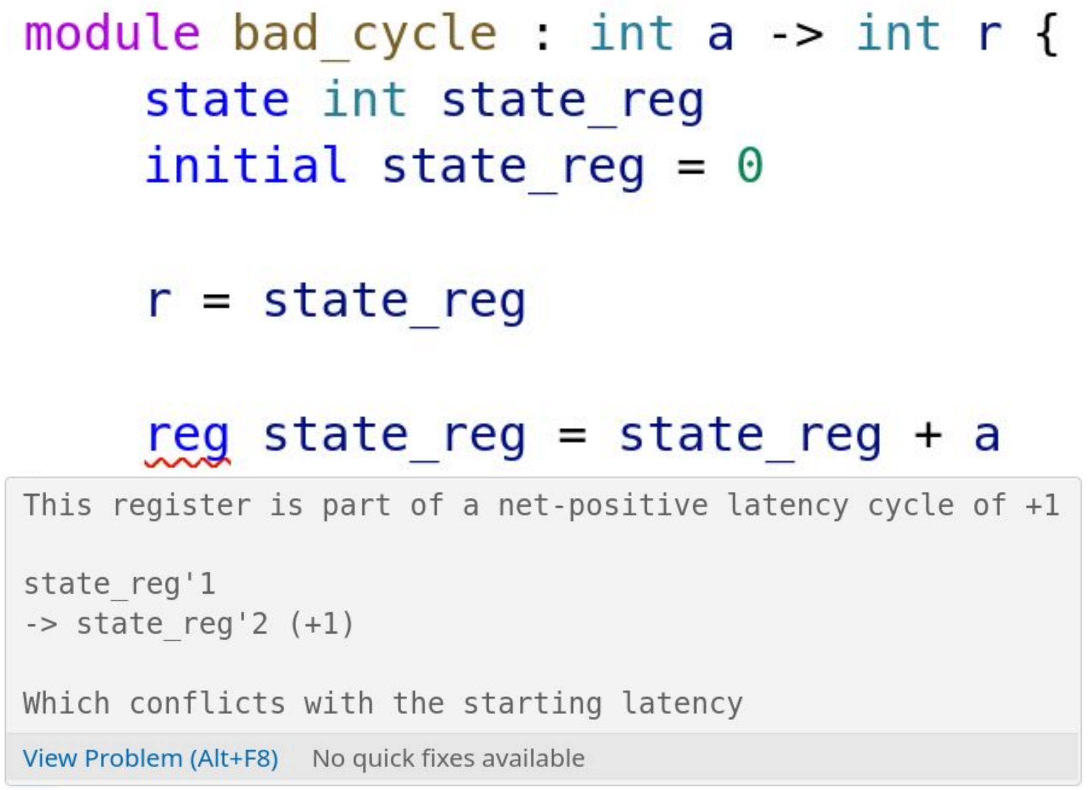
Feedback

LSP

- Errors & Info
- Navigation
- Code Suggestions

Instant In-Editor Feedback

- No separate Compile Step



```
module bad_cycle : int a -> int r {
    state int state_reg
    initial state_reg = 0

    r = state_reg

    reg state_reg = state_reg + a
```

```
This register is part of a net-positive latency cycle of +1

state_reg'1
-> state_reg'2 (+1)

Which conflicts with the starting latency
```

Pragmatism

Let hardware be hardware

Simple things should be simple

Infer where possible

Pragmatism

Let hardware be hardware

Simple things should be simple

Infer where ~~possible~~ beneficial

Examples

XOR gate

```
module xor : bool x1, bool x2 -> bool y {
    bool w1 = !x1
    bool w2 = !x2

    bool w3 = x1 & w2
    bool w4 = x2 & w1

    y = w3 | w4
}
```

Generative Code

```
module matrix_vector_mul :
    int[30][20] mat, int[20] vec -> int[30] result {

    for int row in 0..30 {
        int[20] row_products
        for int col in 0..20 {
            row_products[col] = mat[row][col] * vec[col]
        }
        result[row] = +row_products
    }
}
```

FizzBuzz

```
module fizz_buzz : int v -> int fb {
    gen int FIZZ = 15
    gen int BUZZ = 11
    gen int FIZZ_BUZZ = 1511

    bool fizz = v % 3 == 0
    bool buzz = v % 5 == 0

    if fizz & buzz {
        fb = FIZZ_BUZZ
    } else if fizz {
        fb = FIZZ
    } else if buzz {
        fb = BUZZ
    } else {
        fb = v
    }
}
```

Generative FizzBuzz

```
module fizz_buzz_gen : int v -> int fb {
    gen int FIZZ = 15
    gen int BUZZ = 11
    gen int FIZZ_BUZZ = 1511
    gen int TABLE_SIZE = 256

    gen int[TABLE_SIZE] lut

    for int i in 0..TABLE_SIZE {
        gen bool fizz = i % 3 == 0
        gen bool buzz = i % 5 == 0

        gen int tbl_fb
        if fizz & buzz {
            tbl_fb = FIZZ_BUZZ
        } else if fizz {
            tbl_fb = FIZZ
        } else if buzz {
            tbl_fb = BUZZ
        } else {
            tbl_fb = i
        }

        lut[i] = tbl_fb
    }

    fb = lut[v]
}
```

```
module fizz_buzz_gen : int v -> int fb {
    gen int FIZZ = 15
    gen int BUZZ = 11
    gen int FIZZ_BUZZ = 1511
    gen int TABLE_SIZE = 256

    gen int[TABLE_SIZE] lut

    for int i in 0..TABLE_SIZE {
        gen bool fizz = i % 3 == 0
        gen bool buzz = i % 5 == 0

        gen int tbl_fb
        if fizz & buzz {
            tbl_fb = FIZZ_BUZZ
        } else if fizz {
            tbl_fb = FIZZ
        } else if buzz {
            tbl_fb = BUZZ
        } else {
            tbl_fb = i
        }

        lut[i] = tbl_fb
    }

    fb = lut[v]
}
```

```
gen int TABLE_SIZE = 256
```

```
gen int[TABLE_SIZE] lut
```

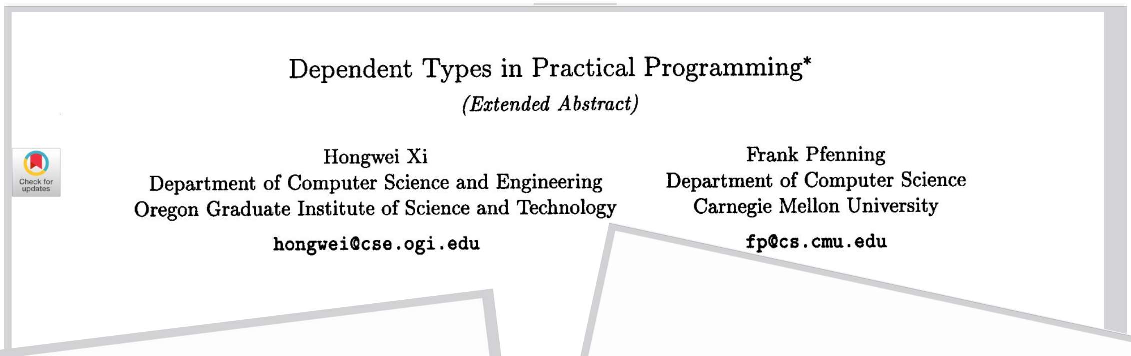
Dependent Types!

ChatGPT

"Dependent types: powerful but often impractical."

Typing

Dependent Types are a nightmare to work with

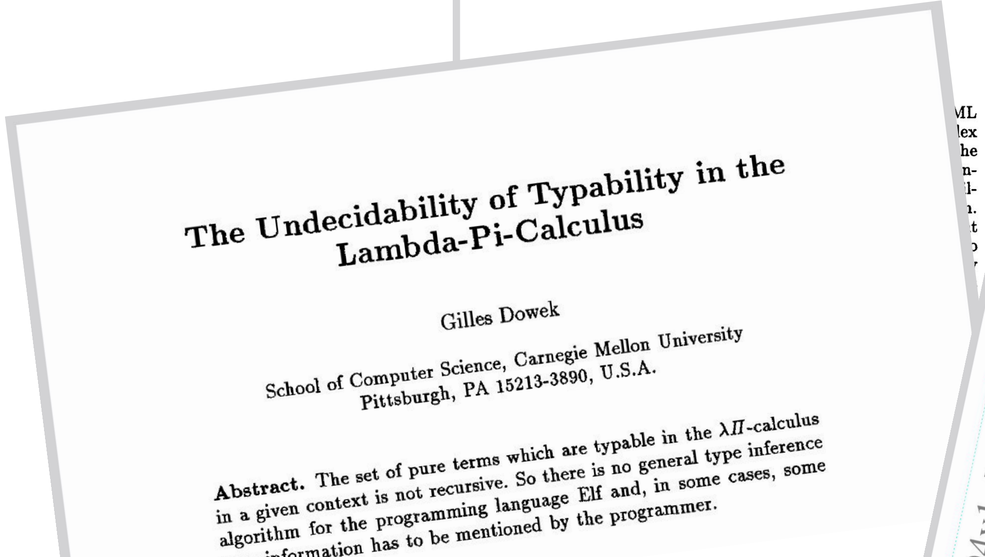


Dependent Types in Practical Programming*

(Extended Abstract)

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

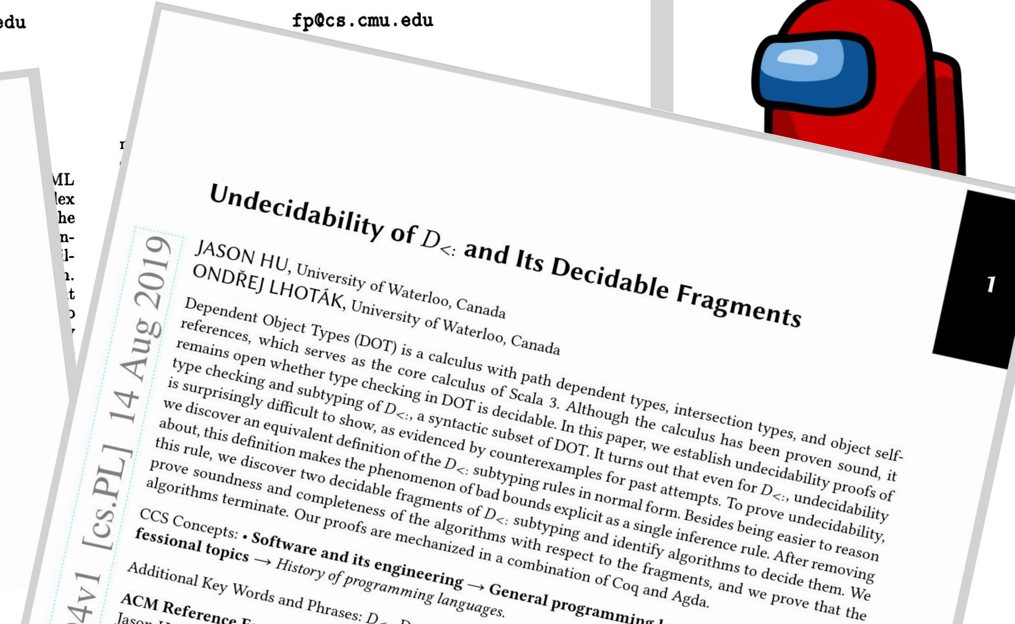


The Undecidability of Typability in the Lambda-Pi-Calculus

Gilles Dowek

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213-3890, U.S.A.

Abstract. The set of pure terms which are typable in the λΠ-calculus in a given context is not recursive. So there is no general type inference algorithm for the programming language Elf and, in some cases, some information has to be mentioned by the programmer.



Undecidability of $D_{<:}$ and Its Decidable Fragments

1

JASON HU, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada

Dependent Object Types (DOT) is a calculus with path dependent types, intersection types, and object self-references, which serves as the core calculus of Scala 3. Although the calculus has been proven sound, it remains open whether type checking in DOT is decidable. In this paper, we establish undecidability proofs of type checking and subtyping of $D_{<:}$, a syntactic subset of DOT. It turns out that even for $D_{<:}$, undecidability is surprisingly difficult to show, as evidenced by counterexamples for past attempts. To prove undecidability, we discover an equivalent definition of the $D_{<:}$ subtyping rules in normal form. Besides being easier to reason about, this definition makes the phenomenon of bad bounds explicit as a single inference rule. After removing this rule, we discover two decidable fragments of $D_{<:}$ subtyping and identify algorithms to decide them. We prove soundness and completeness of the algorithms with respect to the fragments, and we prove that the algorithms terminate. Our proofs are mechanized in a combination of Coq and Agda.

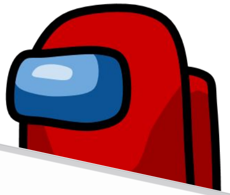
CCS Concepts: • **Software and its engineering** → **General programming** **fessional topics** → *History of programming languages.*

Additional Key Words and Phrases: $D_{<:}$, D

ACM Reference F

Jason H

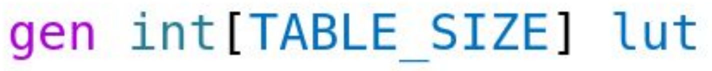
4v1 [cs.PL] 14 Aug 2019



What is the SUS-lution?

- Want simple type checking → use dynamic typing?
  - But do we really need to statically prove *all* parametrizations of a module are “correct”?
  - Or can we simply check each instance?
- Want to use type info to hint user at template level → traits & types?
  - Does this info need to care about array sizes?

⇒ Two typing levels!



```
gen int[TABLE_SIZE] lut
```

Abstract Type

`int[]`

Typecheck at Flattening time

Only type names and structure

LSP Info & Template checking

Concrete Type

`int[256]`

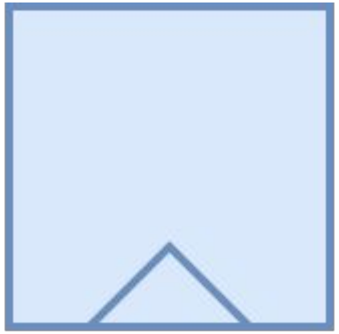
Typecheck at Instantiation time

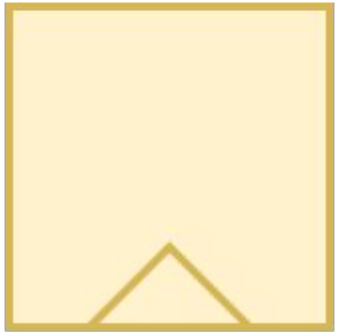
Type names and concrete values

Actually define wires

Latency Counting

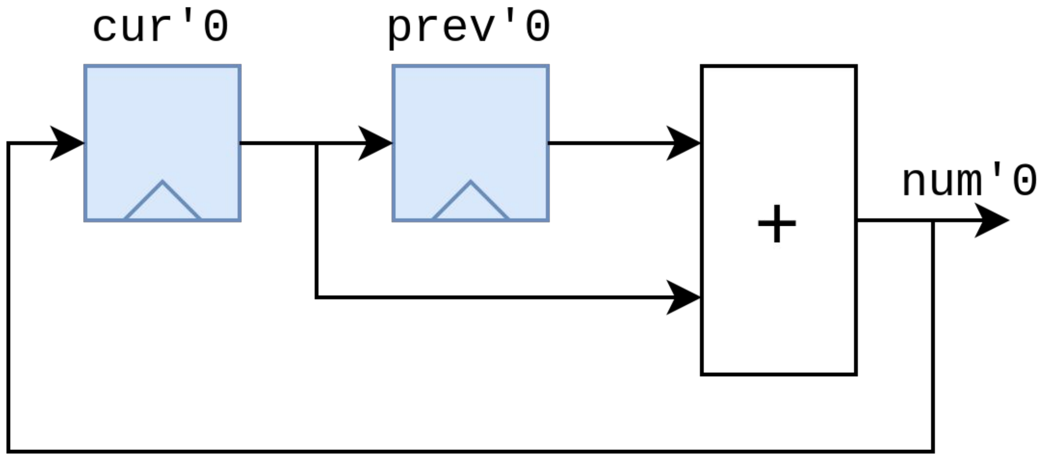
Registers


State


Latency

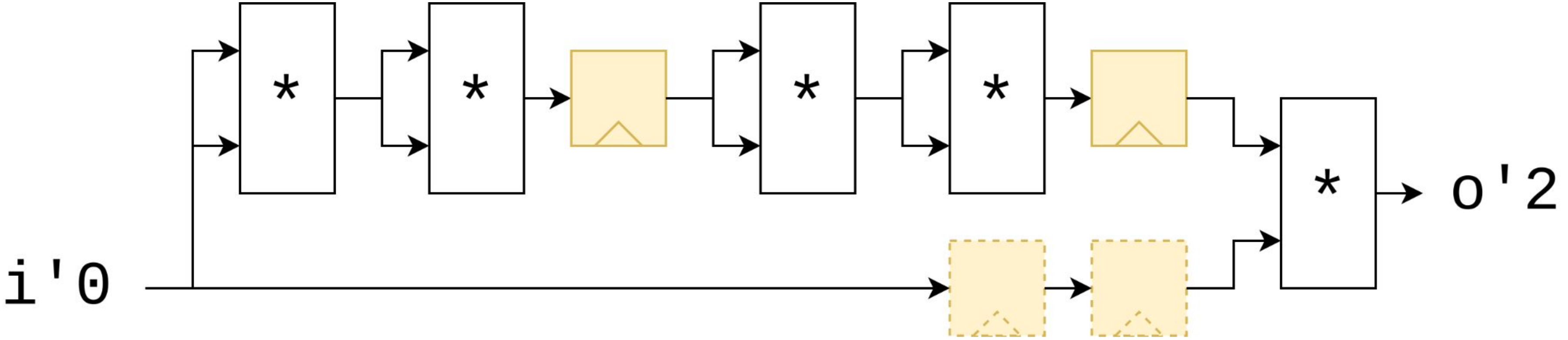
State registers

```
module fibonnaci : -> int num {
    state int cur = 1
    state int prev = 0

    num = cur + prev
    prev = cur
    cur = num
}
```



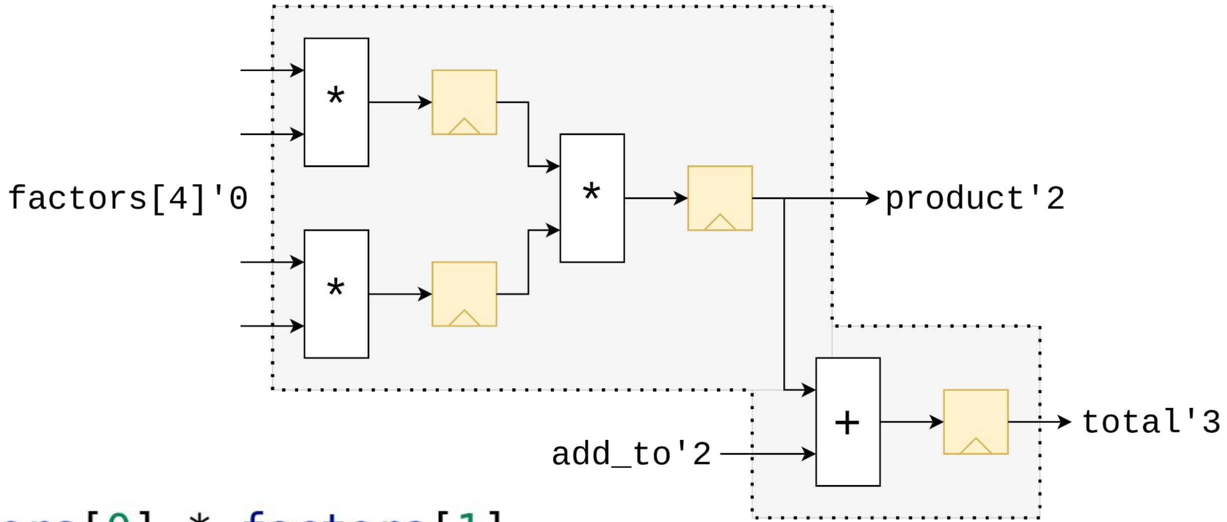
Latency Registers

```
module pow17 : int i -> int o {
        int i2  = i   * i
    reg int i4  = i2  * i2
        int i8  = i4  * i4
    reg int i16 = i8  * i8
            o   = i16 * i
}
```



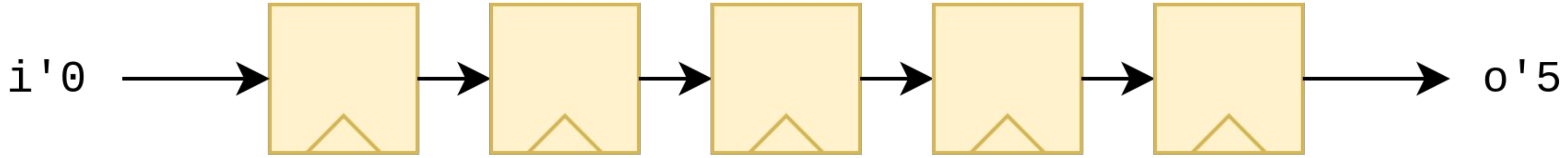
Port Latencies

```
module example_md :
    int[4] factors,
    int add_to ->
    int product,
    int total {

    reg int mul0 = factors[0] * factors[1]
    reg int mul1 = factors[2] * factors[3]

    reg product = mul0 * mul1
    reg total = product + add_to
}
```

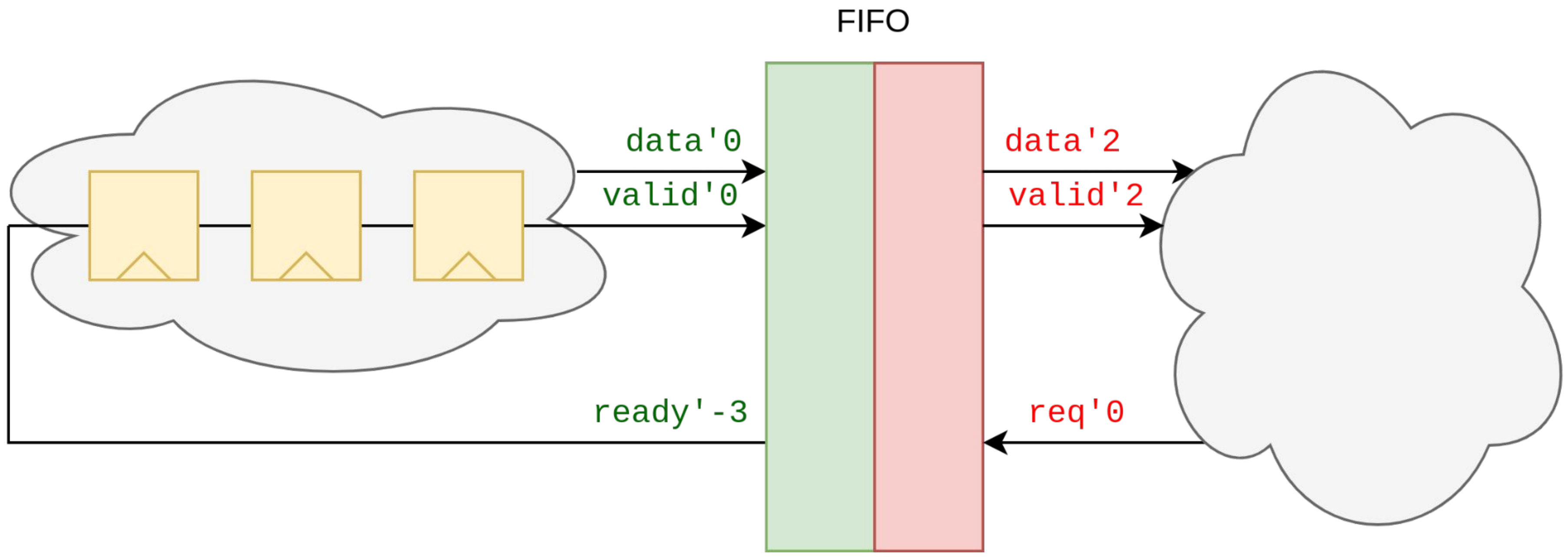


Latency Annotations

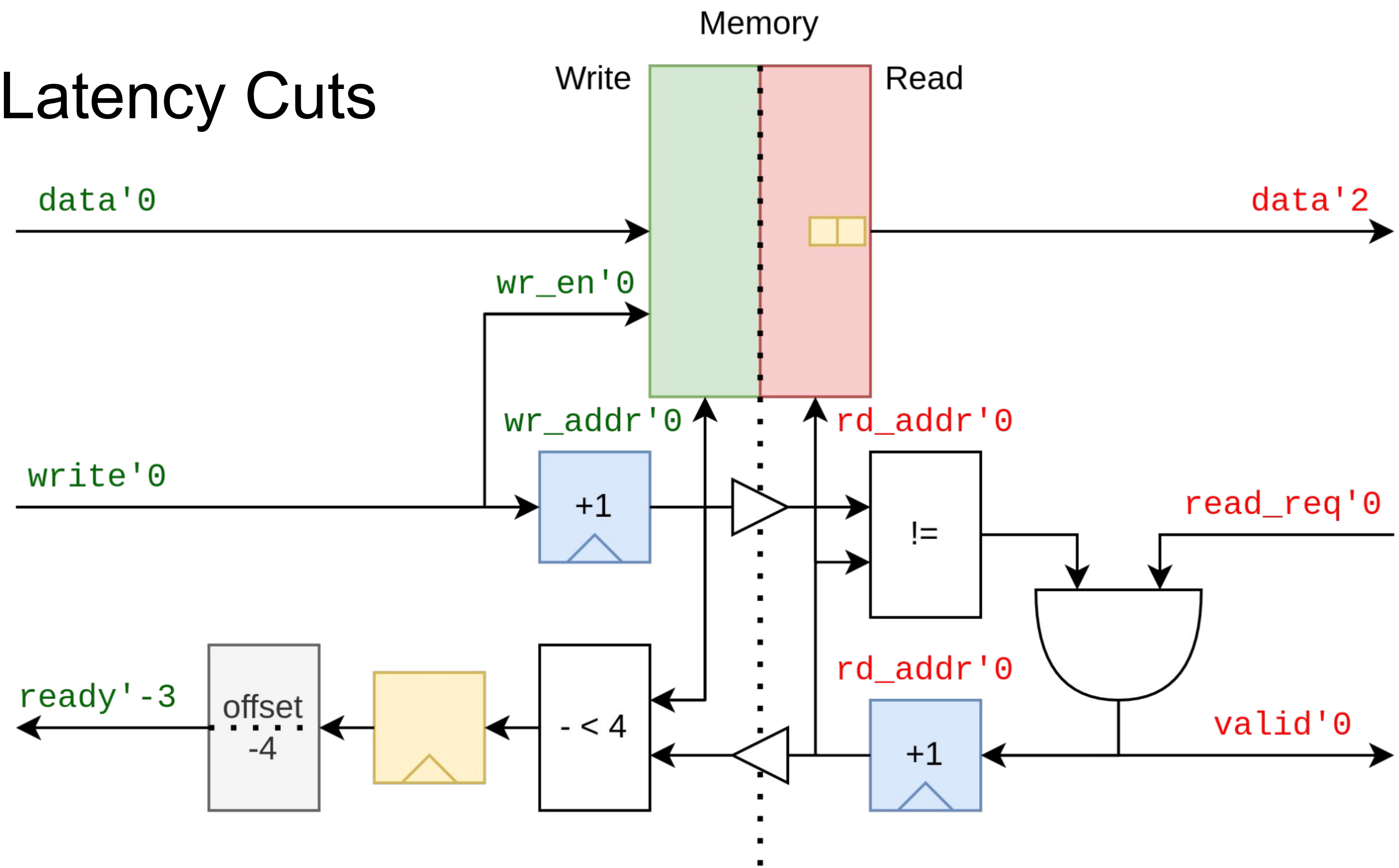
```
module module_taking_time :
    int i'0 -> int o'5 {
    o = i
}
```



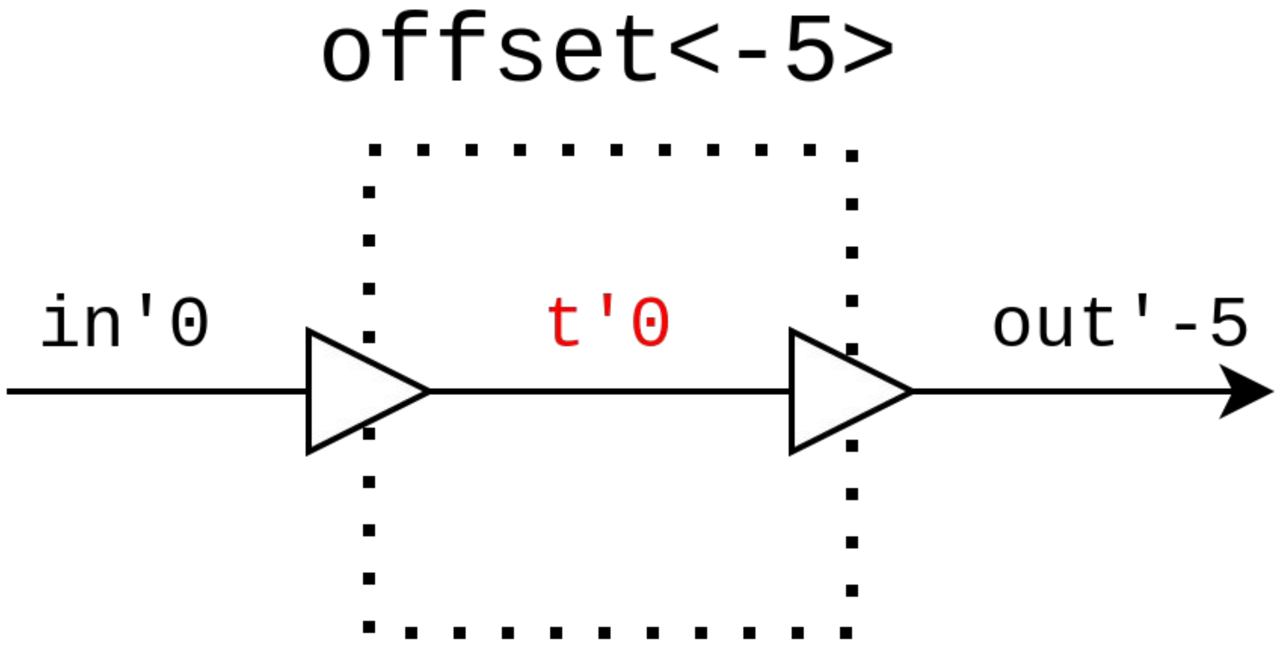
Multiple Interfaces



Latency Cuts

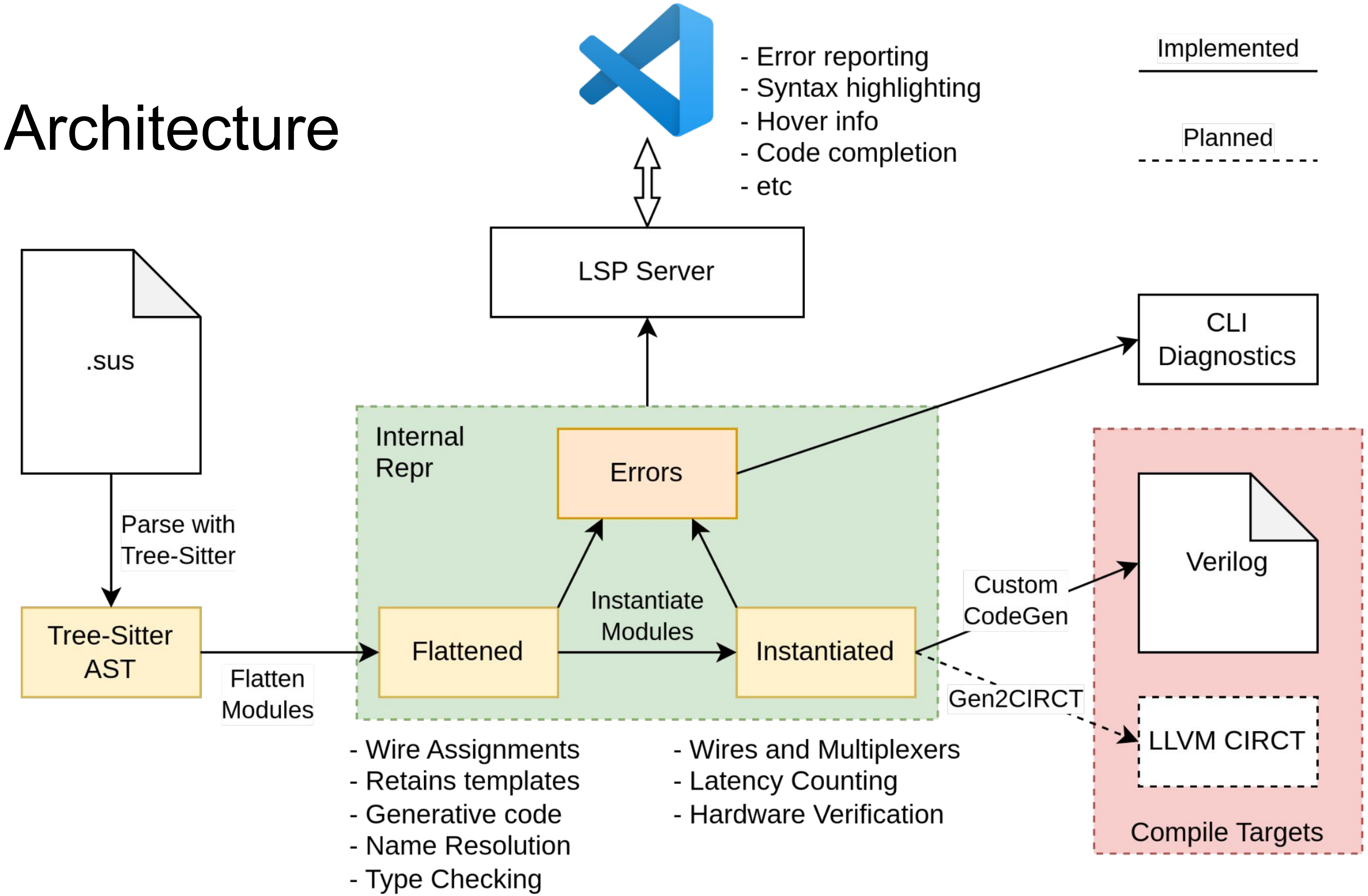


Latency offsets



Architecture & Future

Architecture



Short term milestones
“Build anything in SUS”

- Arbitrary Single-Clock hardware description
  - ~~Migrate from custom parser to Tree-Sitter~~
  - Multi-interface modules
  - Integer Bounds
  - Templates & Generative Parameters
- Standard Library
  - Extern (Verilog) Modules
  - Implement FIFO, Memory Block, Skid Buffer, Distributor, Merger, etc

Vague long term milestones
“SUS stands for Safety”

- CIRCT Compile Target
  - Enables use of btor2 LTL for verification
- Advanced Bounds
  - Use Bounds system as an alternative to Sum Types
  - Protect FIFOs, skid buffers, etc in the STL using LTL verification
- Arbitrary Multi-Clock hardware description
  - STL Clock Domain Crossing modules
  - Multi-Clock STL modules
- Inference of generative parameters
- Custom Types

LSP Demo

github.com/pc2/sus-compiler

- Are “traits” reasonable for hardware?
- C-style types vs Rust-style types?
- Type inference?
- Additional Constructs: `first, only, chain`?
- Backend?
- Abstraction for valid/reset protocols?
- Built-in Verification?
- Bounded integers vs bitwidths?
- Integer representations? (2s cpl, vs 1s cpl vs one-hot vs registered)
- Built-in Floats?
- Sum Types?
- Latency offsets within structs?
- What becomes of SUS after I graduate?