# Specifying Hardware Communication as Programs

**Nikil Shyamsunder**

Ernest Ng  Francis Pham  Adrian Sampson  Kevin Laeufer

Cornell University

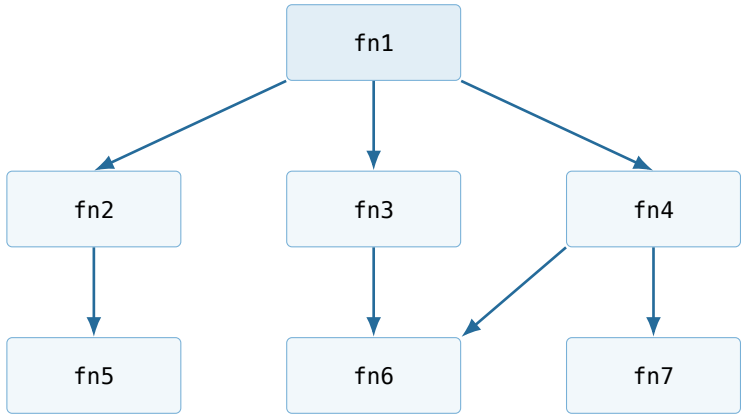




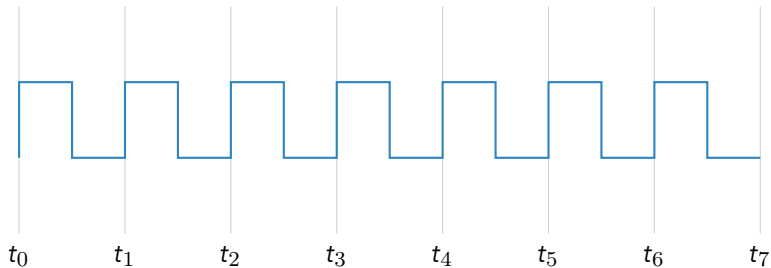
# Calling Functions in Software

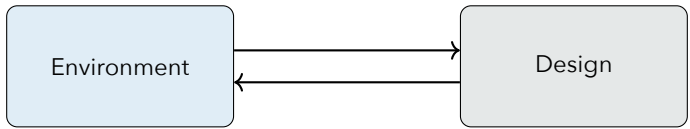
```
li a0, 2
li a1, 3
call add_numbers
// result in a0
```

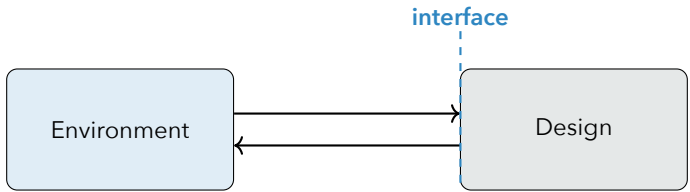
# Agree on a calling convention

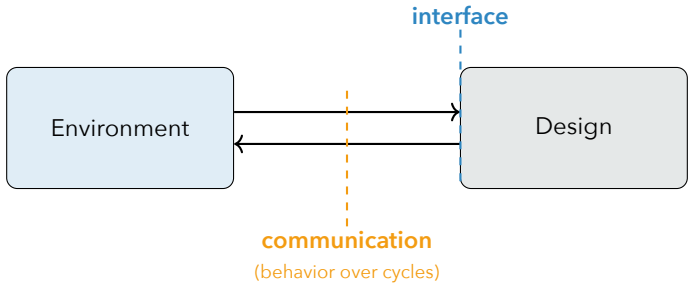


# Hardware is Hard



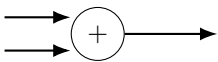

Environment
Design


interface
Environment
Design

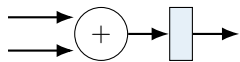

interface
Environment
Design
communication
(behavior over cycles)

# Diversity in communication.

Combinational

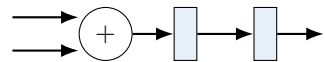


One stage



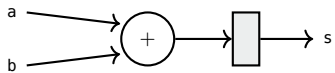
$d = 1 \quad \text{II} = 1$

Two stages



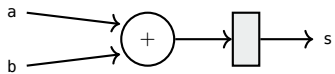
$d = 2 \quad \text{II} = 1$

# Transactions vs. Signals




*Transaction:* “`adding 1 and 2 results in 3`”

# Transactions vs. Signals




*Transaction:* "`adding 1 and 2 results in 3`"

vs. *Signals:* "drive `a=1`, `b=2`; next cycle expect `s=3`"

# Two Kinds of Specification

**Functional Specification**
what the module computes

**Communication Specification**
how and when signals move

# A spectrum of communication protocols

Standardized Buses (Wishbone / AMBA)

Bespoke internal interfaces

Simple handshakes (ready/valid)

# Problem 1: Natural-language Specifications



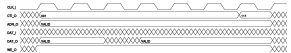


*Illustration 4-8: Constant address burst*

A wishbone master in pipelined mode is shown below.

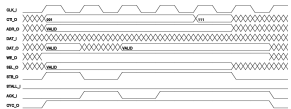


*Illustration 4-9: Constant address burst pipelined mode*

#### *4.4.4 Incrementing Burst Cycle*

An incrementing burst is defined as multiple accesses to consecutive addresses. Each transfer the address is incremented. The increment is dependent on the data array [DAT_O()..[DAT_I()] size; for an 8bit data array the increment is 1, for a 16bit data array the increment is 2, for a 32bit data array the increment is 4, etc.

Increments can be linear or wrapped. Linear increments means the next address is one increment more than the current address. Wrapped increments means that the address increments one, but that the addresses' LSBs are modulo the wrap size.

Table 4-3 Wrap Size address increments

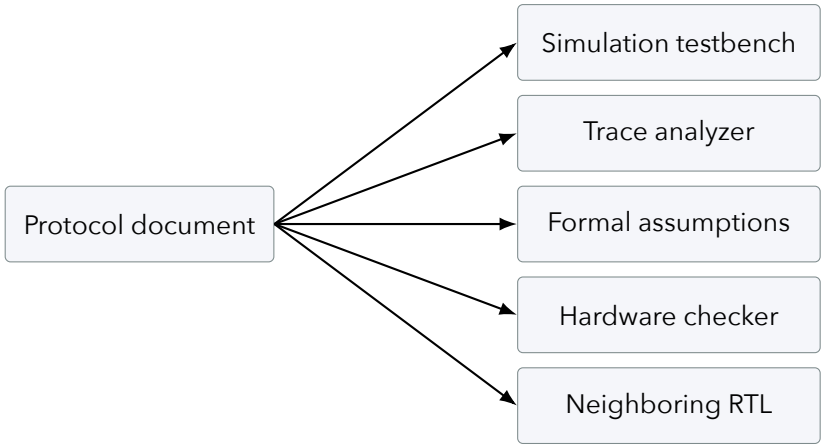
| Starting address' LSBs | Linear | Wrap-4 | Wrap-8 |
|---|---|---|---|
| 000 | 0-1-2-3-4-5-6-7 | 0-1-2-3-4-5-6-7 | 0-1-2-3-4-5-6-7 |
| 001 | 1-2-3-4-5-6-7-8 | 1-2-3-0-5-6-7-4 | 1-2-3-4-5-6-7-0 |
| 010 | 2-3-4-5-6-7-8-9 | 2-3-0-1-6-7-4-5 | 2-3-4-5-6-7-0-1 |
| 011 | 3-4-5-6-7-8-9-A | 3-0-1-2-7-4-5-6 | 3-4-5-6-7-0-1-2 |
| 100 | 4-5-6-7-8-9-A-B | 4-5-6-7-8-9-A-B | 4-5-6-7-0-1-2-3 |
| 101 | 5-6-7-8-9-A-B-C | 5-6-7-4-9-A-B-8 | 5-6-7-0-1-2-3-4 |
| 110 | 6-7-8-9-A-B-C-D | 6-7-4-5-A-B-8-9 | 6-7-0-1-2-3-4-5 |
| 111 | 7-8-9-A-B-C-D-E | 7-4-5-6-B-8-9-A | 7-0-1-2-3-4-5-6 |

Example: Processor cache line read

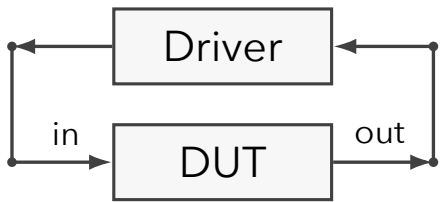
 Copyright © 2010 OpenCores 

OpenCores, WISHBONE B4, incrementing-address burst cycles

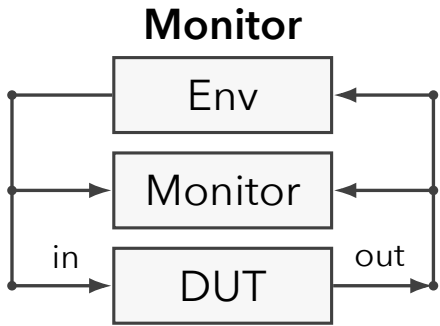
# Ad Hoc Interpretations



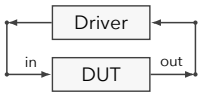
# Driver



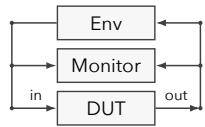
# Monitor



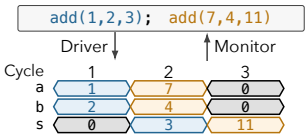
**Driver**




**Monitor**




**Interpretation**



## What should a communication specification provide?

Easily understood by humans

# What should a communication specification provide?

Easily understood by humans

Formal semantics

## What should a communication specification provide?

Easily understood by humans

Formal semantics

Driver interpretation

## What should a communication specification provide?

Easily understood by humans

Formal semantics

Driver interpretation

Monitor interpretation

## What should a communication specification provide?

| | Natural language |
|---|---|
| Easily understood by humans | ✓✓ |
| Formal semantics | |
| Driver interpretation | |
| Monitor interpretation | |

# What should a communication specification provide?

| | Natural language | SVA / PSL |
|---|---|---|
| Easily understood by humans | ✓✓ | |
| Formal semantics | | ✓ |
| Driver interpretation | | |
| Monitor interpretation | | ✓* |

*Temporal-property checking; transaction reconstruction needs additional structure.

# What should a communication specification provide?

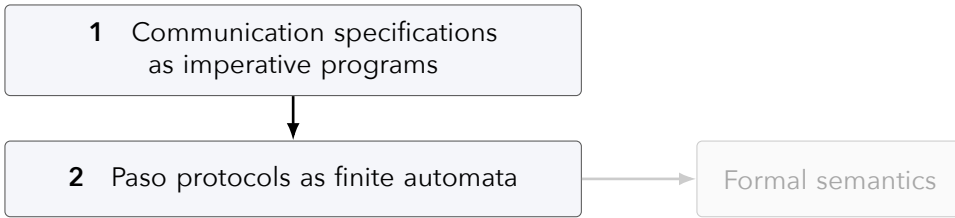
| | Natural language | SVA / PSL | Paso |
|---|---|---|---|
| Easily understood by humans | ✓✓ | | ✓ |
| Formal semantics | | ✓ | ✓ |
| Driver interpretation | | | ✓ |
| Monitor interpretation | | ✓* | ✓ |

*Temporal-property checking; transaction reconstruction needs additional structure.

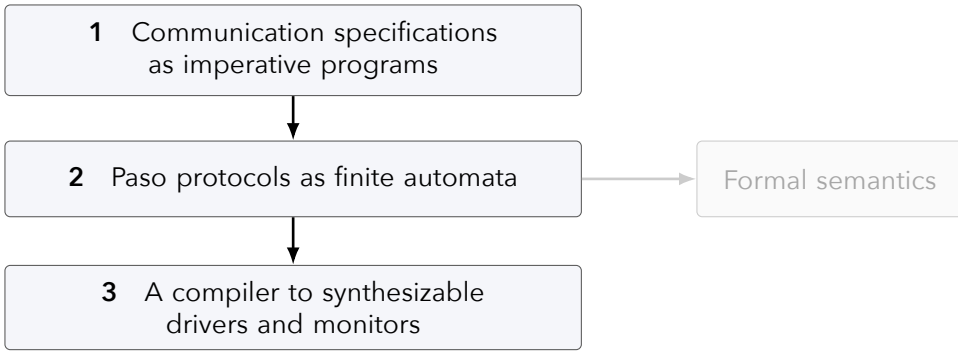
# Talk Outline

**1** Communication specifications as imperative programs

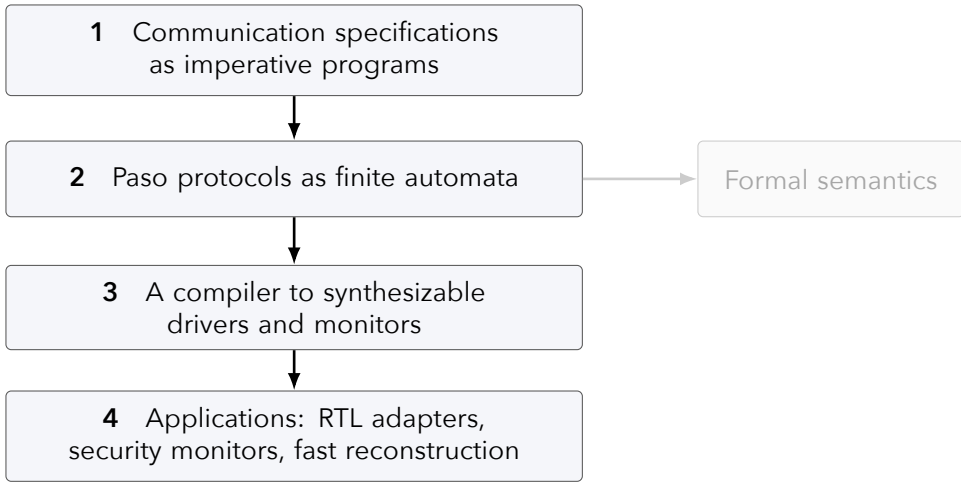
# Talk Outline



# Talk Outline

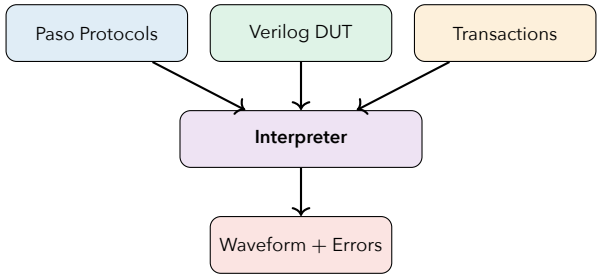


# Talk Outline

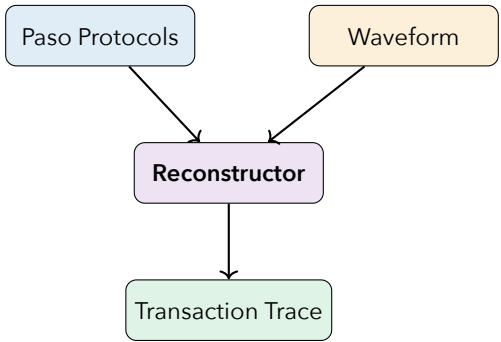


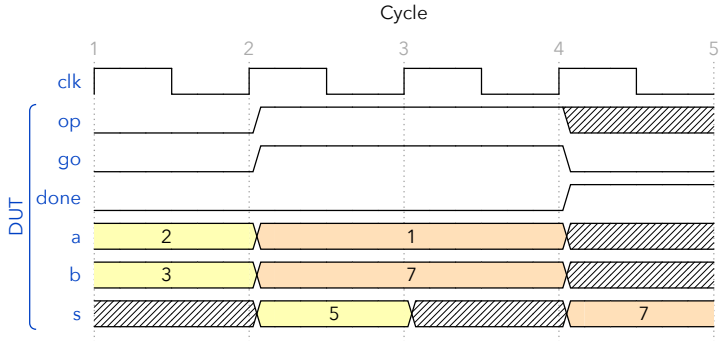
# Communication Specifications as Imperative Programs

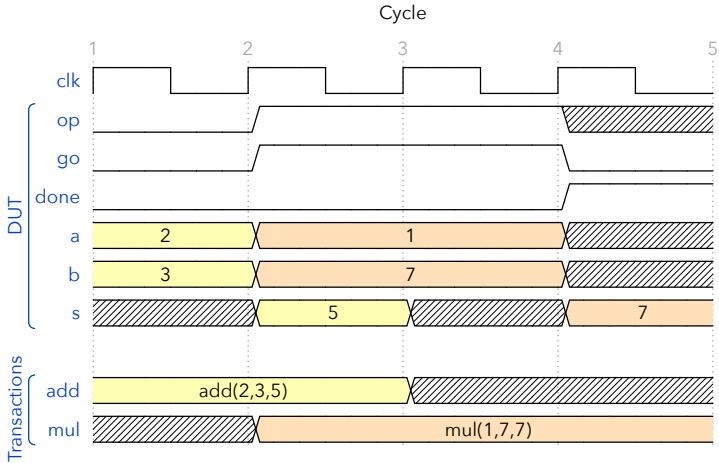
# Interpreter



# Reconstructor: Motivation



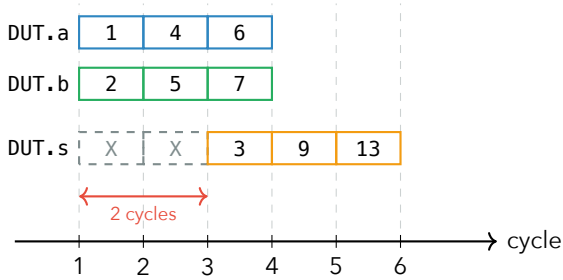

Cycle
1
2
3
4
5
clk
op
go
done
a
b
s
DUT
2
1
3
7
5
7


Cycle
1
2
3
4
5
clk
DUT
op
go
done
a
2
1
b
3
7
s
5
7
Transactions
add
add(2,3,5)
mul
mul(1,7,7)

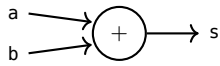
## Latency-Sensitive Interface

Fixed timing relationship between input and output

# Running Example: Depth-2 Pipelined Adder



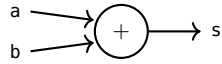
# A Protocol Header



# A Protocol Header

```
prot add_one_two<DUT: Adder>() {

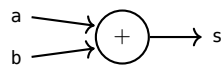
}
```



# A Protocol Header

```
prot add_one_two<DUT: Adder>() {

}
```

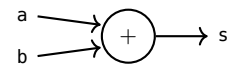



Call: `add_one_two()`

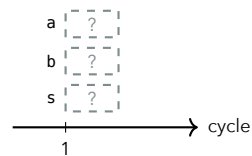
# A Protocol Header

```
prot add_one_two<DUT: Adder>() {

}
```

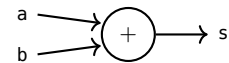



Call: `add_one_two()`

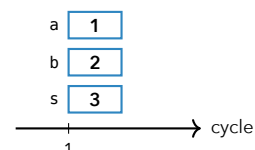


# Driving Inputs

```
prot add_one_two<DUT: Adder>() {
  DUT.a := 1;
  DUT.b := 2;
}
```



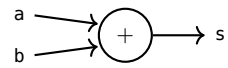

Call: `add_one_two()`



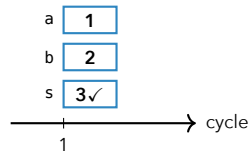

no local variables

# Checking Outputs

```
prot add_one_two<DUT: Adder>() {
  DUT.a := 1;
  DUT.b := 2;
  assert_eq(DUT.s, 3);
}
```

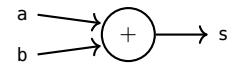



Call: `add_one_two()`

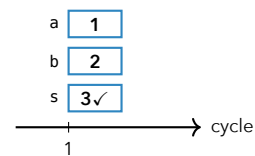


# Parameterizing the Protocol

```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  assert_eq(DUT.s, s);
}
```



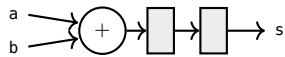

Call: `add(1, 2, 3)`




Parameters are immutable

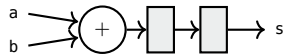
# `step()` – Sequential Circuits

```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  step();
  assert_eq(DUT.s, s);
}
```



# `step()` – Sequential Circuits

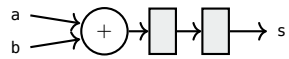
```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  step();
  assert_eq(DUT.s, s);
}
```



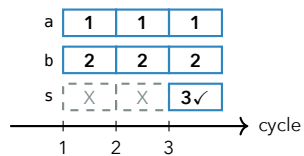

Call: `add(1, 2, 3)`

# `step()` – Sequential Circuits

```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  step();
  assert_eq(DUT.s, s);
}
```

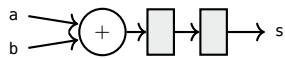



Call: `add(1, 2, 3)`



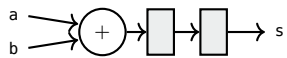
# `fork()` – Pipelining Transactions

```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  fork();
  step();
  assert_eq(DUT.s, s);
}
```



# `fork()` – Pipelining Transactions

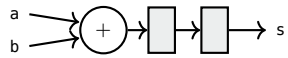
```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  fork();
  step();
  assert_eq(DUT.s, s);
}
```



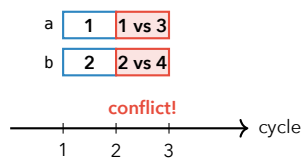

Call: `add(1,2,3); add(3,4,7)`

# `fork()` – Pipelining Transactions

```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  fork();
  step();
  assert_eq(DUT.s, s);
}
```




Call: `add(1,2,3); add(3,4,7)`



# DontCare – Releasing Ports

```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  DUT.a := X; DUT.b := X;
  fork();
  step();
  assert_eq(DUT.s, s);
}
```

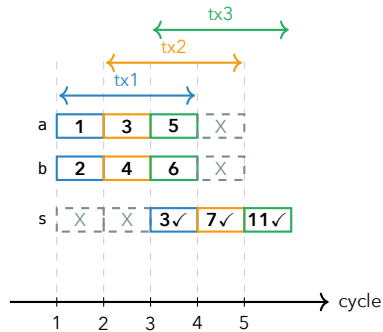
- `X` (DontCare): this port is no longer meaningful for my transaction

# Pipelined Adder

Transaction trace:

```
add(1,2,3); add(3,4,7); add(5,6,11);
```

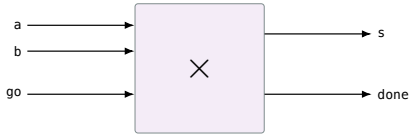
```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  DUT.a := X;
  DUT.b := X;
  fork();
  step();
  assert_eq(DUT.s, s);
}
```

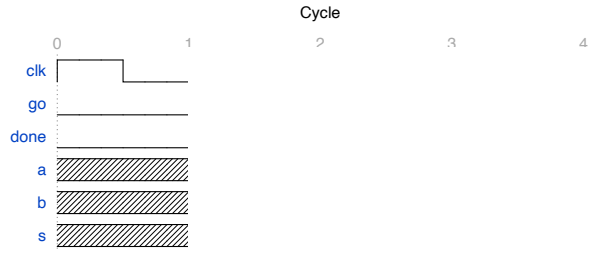


## Latency-Insensitive Interface

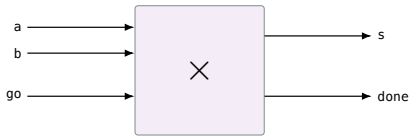
Explicit synchronization signals coordinate data transfer

# Variable latency multiplier: go-done

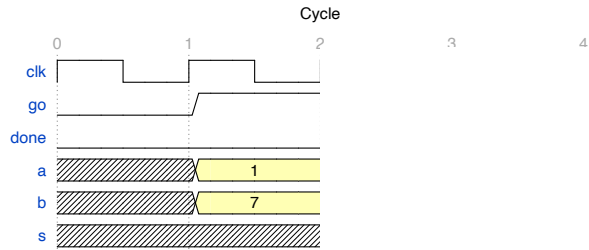


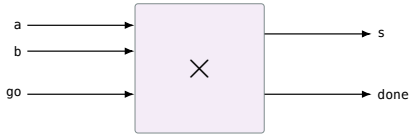
# Variable latency multiplier: go-done



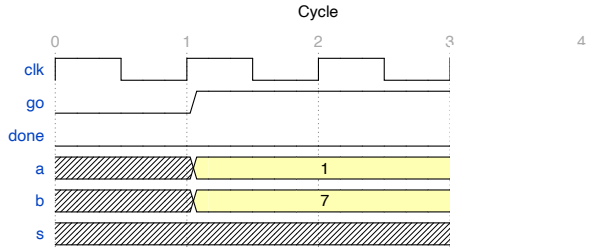

1. Supply operands and raise `go`



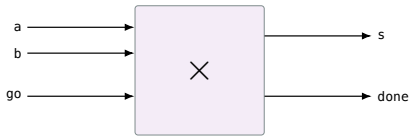
# Variable latency multiplier: go-done



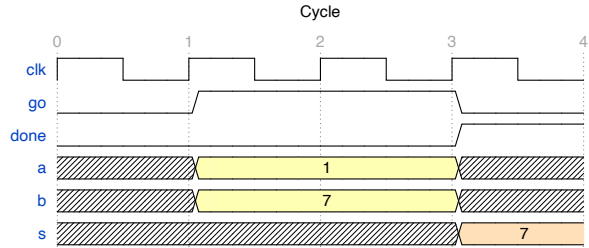

1. Supply operands and raise `go`
2. Hold `go` and operands while `done=0`



# Variable latency multiplier: go-done




1. Supply operands and raise `go`
2. Hold `go` and operands while `done=0`
3. DUT raises `done` when the result is available
4. Sample the result and lower `go` in that cycle



# Go-done (SVA)

```
property calyx_go_done;
  @(posedge clk) disable iff (reset)
    // Idle or waiting for completion
    (!dut.go || (dut.go && !dut.done))
    and
    // From waiting: complete, or keep waiting with stable inputs
    ((dut.go && !dut.done) |=>
      (dut.done
        ? !dut.go
        : (dut.go && $stable({dut.a, dut.b}))));
endproperty

assert property (calyx_go_done);
```

```
prot mul<D: Multiplier>(
  a: u32, b: u32, s: u32
) {
  D.a := a; D.b := b;
  D.go := 1;
  // ...
}
```

1. Supply operands and raise `go`

```
prot mul<D: Multiplier>(
  a: u32, b: u32, s: u32
) {
  D.a := a; D.b := b;
  D.go := 1;
  while !D.done { step(); }
  // ...
}
```

2. Hold `go` and operands while `done=0`

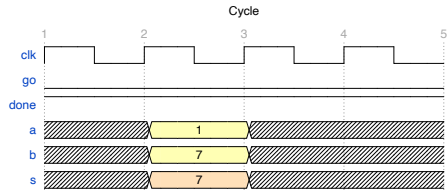
```
prot mul<D: Multiplier>(
  a: u32, b: u32, s: u32
) {
  D.a := a; D.b := b;
  D.go := 1;
  while !D.done { step(); }
  D.a := X; D.b := X;
  D.go := 0;
  assert_eq(D.s, s);
  step();
}
```

3. DUT raises **done** when the result is available

4. Sample the result and lower **go** in that cycle

# Paso is sequential

```
prot mul<D: Multiplier>(
  a: u32, b: u32, s: u32
) {
  D.a := a; D.b := b;
  D.go := 1;
  while !D.done { step(); }
  D.a := X; D.b := X;
  D.go := 0;
  assert_eq(D.s, s);
  step();
}
```



# Sequence parameters

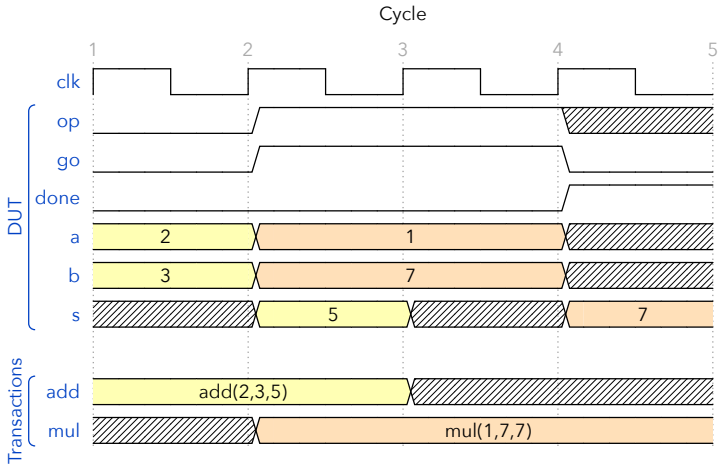
```
prot write_burst<W: Wishbone>(base: u32, data: [u32]++) {
  // Burst setup omitted
  for d in data {
    W.ADR := base + (iter_count::<u30>() ## 2'b00);
    W.DAT_O := d;
    if is_last() { W.CTI := 3'b111; }
    while !W.ACK { step(); }
    assert_eq(W.ERR || W.RTY, 0);
  }
}
```

# Multiple Protocols

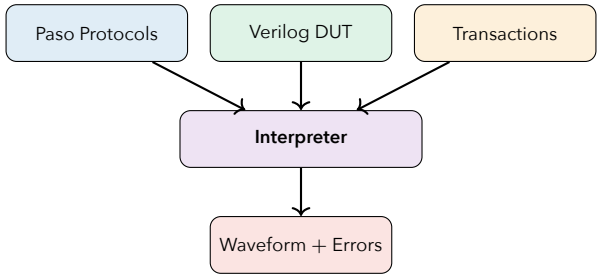
```
prot add<D: ALU>(a: u32, b: u32, s: u32) {
  D.a := a; D.b := b;
  D.op := 0; D.go := 0;
  step();
  D.a := X; D.b := X;
  D.op := X; D.go := X;
  assert_eq(D.s, s);
  fork(); step();
}
```

```
prot mul<D: ALU>(a: u32, b: u32, s: u32) {
  D.a := a; D.b := b;
  D.op := 1; D.go := 1;
  while !D.done { step(); }
  D.a := X; D.b := X;
  D.go := 0; D.op := X;
  assert_eq(D.s, s);
  step();
}
```

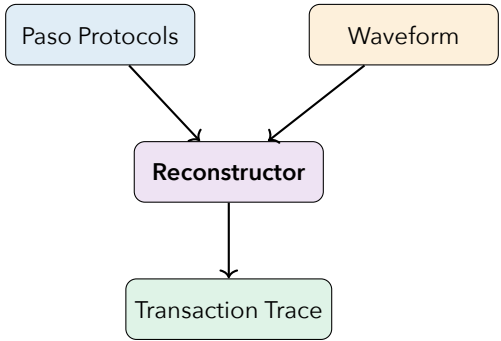
# Add, then a two-cycle multiplication



# Interpreter



# Reconstructor: Motivation

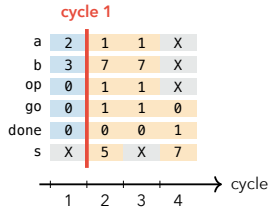


# Reconstruction in software

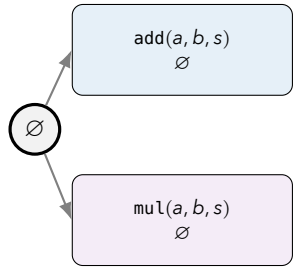
- Turn protocol statements into constraints on parameters
- Explore possible protocol starts breadth first
- Branch at most once per cycle

# Reconstruction, step by step

```
prot add(a,b,s) {
>  DUT.a:=a;
   DUT.b:=b;
   DUT.op:=0; DUT.go:=0;
   step();
   DUT.a:=X; DUT.b:=X;
   DUT.op:=X; DUT.go:=X;
   assert_eq(DUT.s,s);
   fork(); step();
}
```

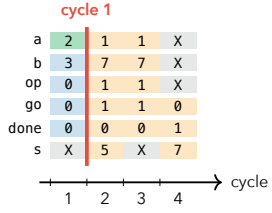



Exploration tree

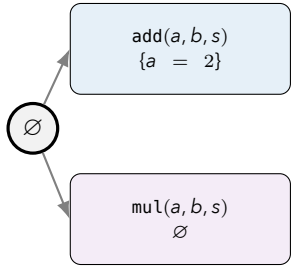


# Reconstruction, step by step

```
prot add(a,b,s) {
>  DUT.a:=a;
   DUT.b:=b;
   DUT.op:=0; DUT.go:=0;
   step();
   DUT.a:=X; DUT.b:=X;
   DUT.op:=X; DUT.go:=X;
   assert_eq(DUT.s,s);
   fork(); step();
}
```

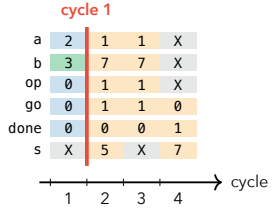



Exploration tree

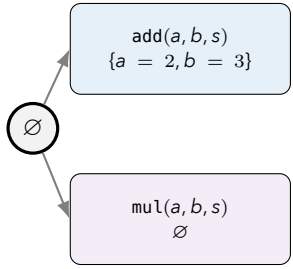


# Reconstruction, step by step

```
prot add(a,b,s) {
  DUT.a:=a;
> DUT.b:=b;
  DUT.op:=0; DUT.go:=0;
  step();
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X; DUT.go:=X;
  assert_eq(DUT.s,s);
  fork(); step();
}
```

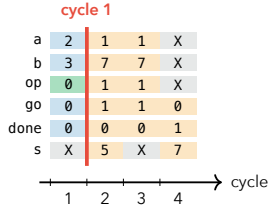



## Exploration tree

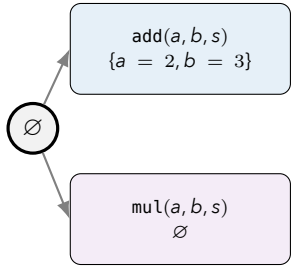


# Reconstruction, step by step

```
prot add(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
  DUT.op:=0; DUT.go:=0;
>  step();
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X; DUT.go:=X;
  assert_eq(DUT.s,s);
  fork(); step();
}
```

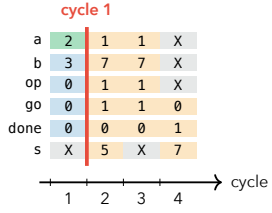



Exploration tree

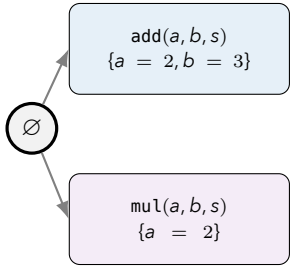


# Reconstruction, step by step

```
prot mul(a,b,s) {
>  DUT.a:=a;
   DUT.b:=b;
   DUT.op:=1; DUT.go:=1;
   step();
   while !DUT.done {
     step();
   }
   DUT.go:=0;
   DUT.a:=X; DUT.b:=X;
   DUT.op:=X;
   assert_eq(DUT.s,s);
   step();
}
```

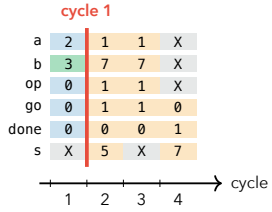



Exploration tree

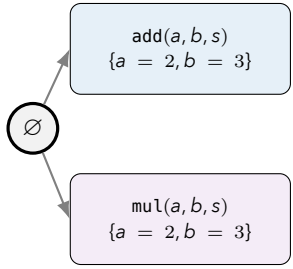


# Reconstruction, step by step

```
prot mul(a,b,s) {
  DUT.a:=a;
> DUT.b:=b;
  DUT.op:=1; DUT.go:=1;
  step();
  while !DUT.done {
    step();
  }
  DUT.go:=0;
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X;
  assert_eq(DUT.s,s);
  step();
}
```

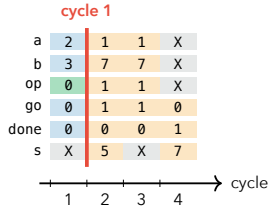



Exploration tree

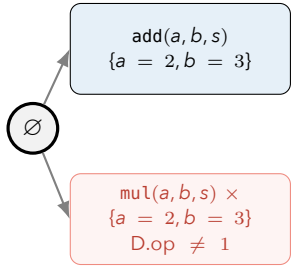


# Reconstruction, step by step

```
prot mul(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
>  DUT.op:=1; DUT.go:=1;
  step();
  while !DUT.done {
    step();
  }
  DUT.go:=0;
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X;
  assert_eq(DUT.s,s);
  step();
}
```

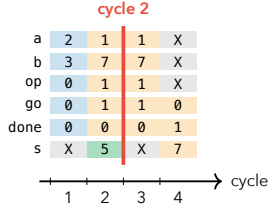



## Exploration tree

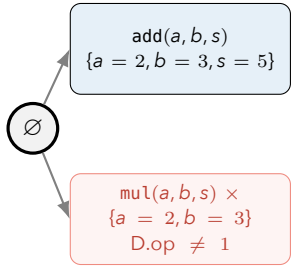


# Reconstruction, step by step

```
prot add(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
  DUT.op:=0; DUT.go:=0;
  step();
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X; DUT.go:=X;
>  assert_eq(DUT.s,s);
  fork(); step();
}
```




## Exploration tree

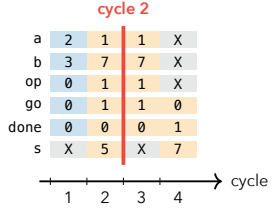



## Inferred trace

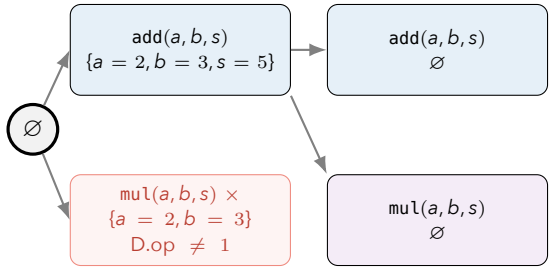
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot add(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
  DUT.op:=0; DUT.go:=0;
  step();
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X; DUT.go:=X;
  assert_eq(DUT.s,s);
> fork(); step();
}
```




## Exploration tree

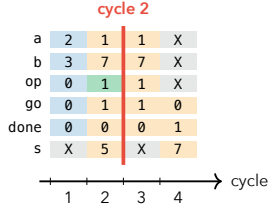



## Inferred trace

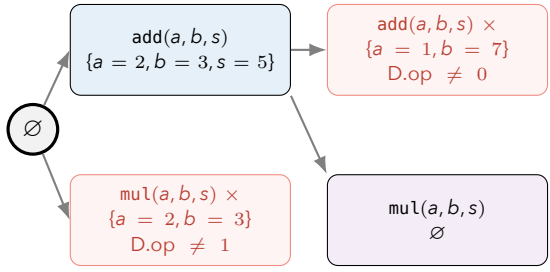
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot add(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
>  DUT.op:=0; DUT.go:=0;
  step();
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X; DUT.go:=X;
  assert_eq(DUT.s,s);
  fork(); step();
}
```




## Exploration tree

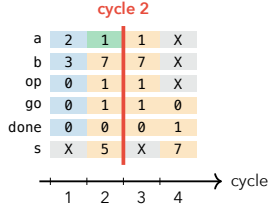



## Inferred trace

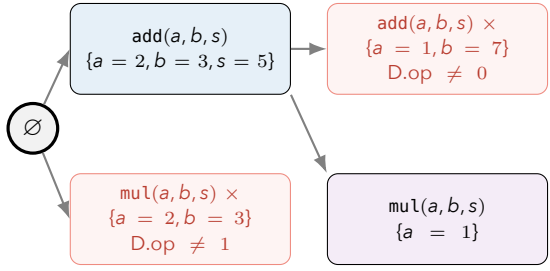
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot mul(a,b,s) {
>  DUT.a:=a;
   DUT.b:=b;
   DUT.op:=1; DUT.go:=1;
   step();
   while !DUT.done {
     step();
   }
   DUT.go:=0;
   DUT.a:=X; DUT.b:=X;
   DUT.op:=X;
   assert_eq(DUT.s,s);
   step();
}
```




## Exploration tree

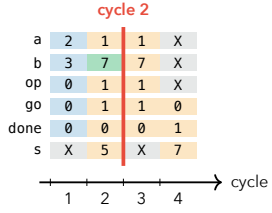



## Inferred trace

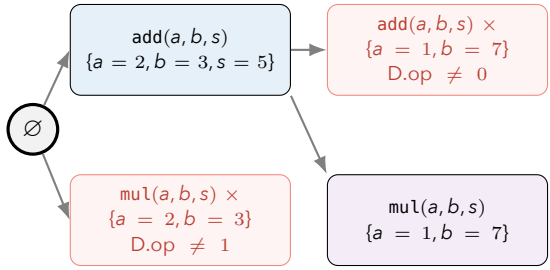
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot mul(a,b,s) {
  DUT.a:=a;
>  DUT.b:=b;
  DUT.op:=1; DUT.go:=1;
  step();
  while !DUT.done {
    step();
  }
  DUT.go:=0;
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X;
  assert_eq(DUT.s,s);
  step();
}
```




## Exploration tree

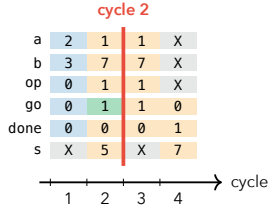



## Inferred trace

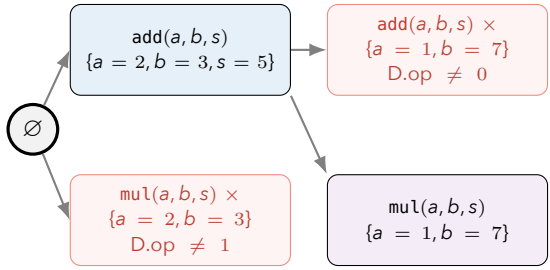
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot mul(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
  DUT.op:=1; DUT.go:=1;
>  step();
  while !DUT.done {
    step();
  }
  DUT.go:=0;
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X;
  assert_eq(DUT.s,s);
  step();
}
```




## Exploration tree

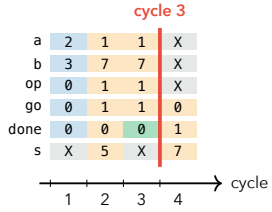



## Inferred trace

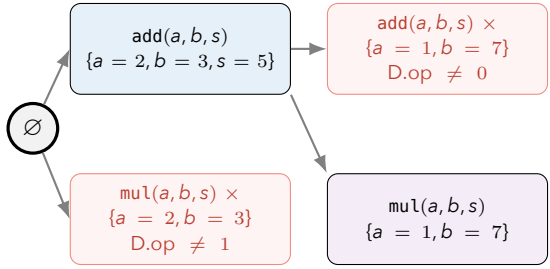
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot mul(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
  DUT.op:=1; DUT.go:=1;
  step();
  while !DUT.done {
>   step();
  }
  DUT.go:=0;
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X;
  assert_eq(DUT.s,s);
  step();
}
```




## Exploration tree

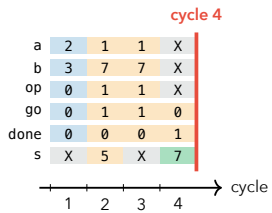



## Inferred trace

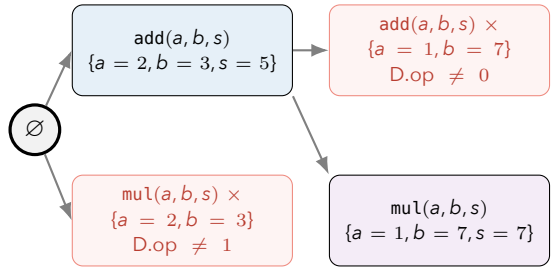
```
add(2,3,5)
```

# Reconstruction, step by step

```
prot mul(a,b,s) {
  DUT.a:=a;
  DUT.b:=b;
  DUT.op:=1; DUT.go:=1;
  step();
  while !DUT.done {
    step();
  }
  DUT.go:=0;
  DUT.a:=X; DUT.b:=X;
  DUT.op:=X;
> assert_eq(DUT.s,s);
  step();
}
```




## Exploration tree




## Inferred trace

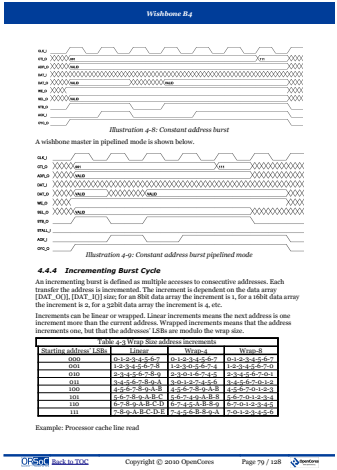
```
add(2,3,5)
mul(1,7,7)
```

# Language design

Restrictions keep reconstruction lightweight

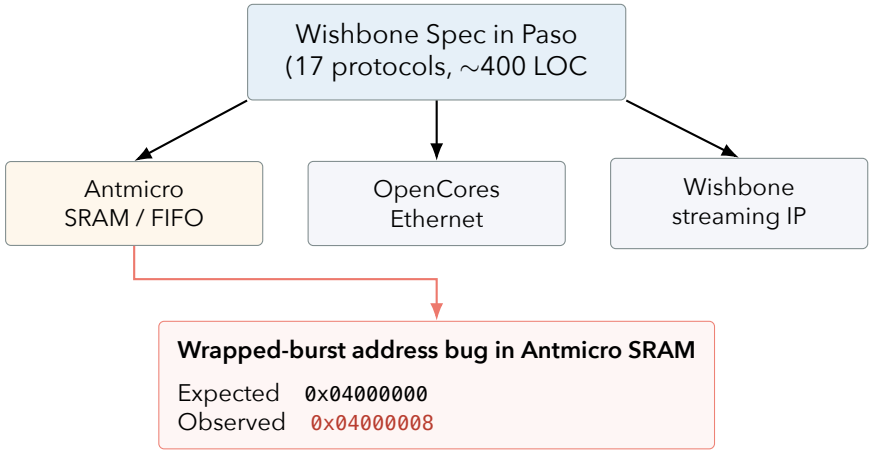
- No SMT solver
- No backtracking over parameter values
- One branching point per cycle limits search-tree growth

# Evaluating Paso in software




```
prot write_burst_increment_linear<DUT: Wishbone>(
    mask: u4, start_addr: u32, data: [u32]++) {
  DUT.RST := 1'b0;
  DUT.CTI := 3'b010;
  DUT.BTE := 2'b00;
  DUT.CYC := 1'b1;
  DUT.STB := 1'b1;
  DUT.WE := 1'b1;
  DUT.SEL := mask;
  for d in data {
    DUT.ADR := start_addr + (iter_count::<u30>() ## 2'b00);
    DUT.DAT_O := d;
    if is_last() {
      DUT.CTI := 3'b111;
    }
    while DUT.ACK == 1'b0 {
      step();
    }
    step();
  }
}
```

# Evaluating Paso in software



# What is still missing?

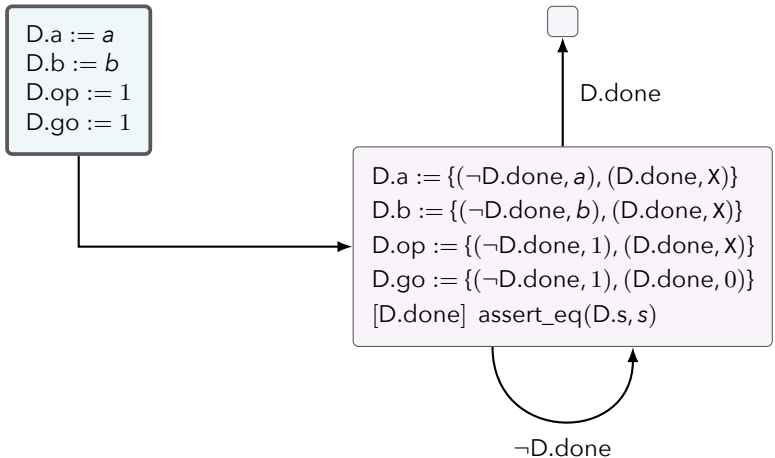
| | |
|---|---|
| Readable protocol | ✓ |
| Driver and monitor interpretations | ✓ |
| Formal semantics | next |
| Synthesizable drivers and monitors | next |

# Protocols as finite automata

# Why compile protocols to hardware?

- Hardware runtime monitors for security
- Adapters between HLS compilers and legacy RTL
  An FFI for hardware

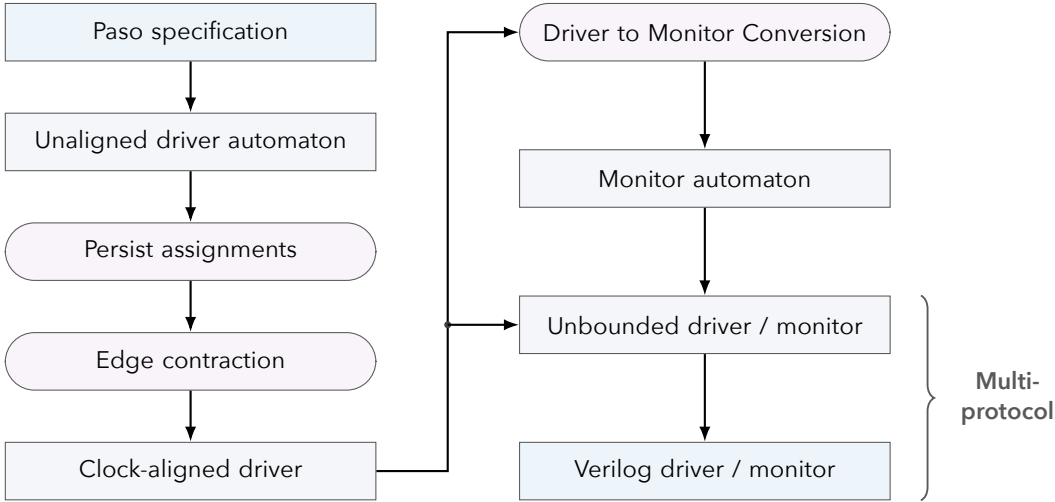
# Single-transaction multiplier driver



# Challenges

- Hardware is inherently parallel
- Hardware is inherently finite

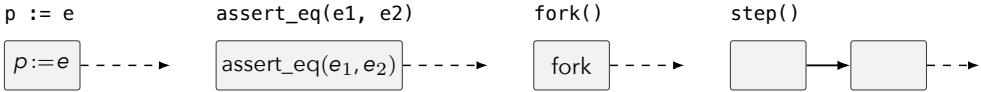
# The compiler pipeline



# Single-protocol, single-transaction compilation

# Lowering to automata

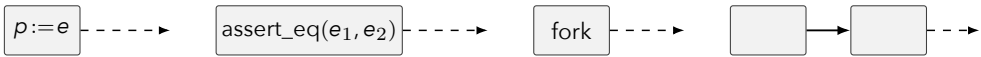
### Atomic statements



# Lowering to automata

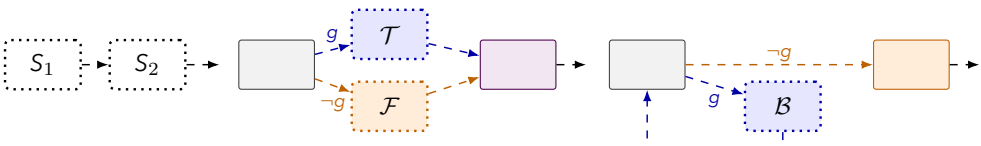
## Atomic statements

`p := e` `assert_eq(e1, e2)` `fork()` `step()`




## Recursive statements

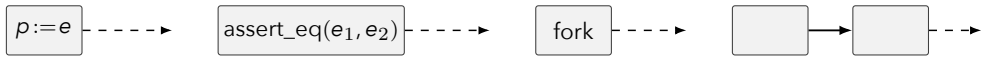
$S_1$; $S_2$ `if g { T } else { F }` `while g { B }`



# Lowering to automata

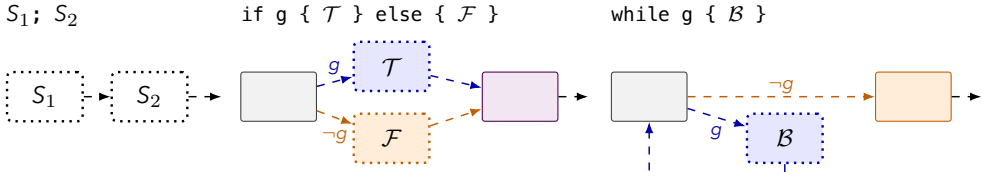
## Atomic statements

`p := e` `assert_eq(e1, e2)` `fork()` `step()`



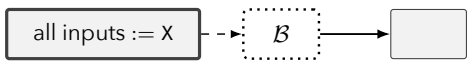

## Recursive statements

$S_1$; $S_2$ `if g { T } else { F }` `while g { B }`

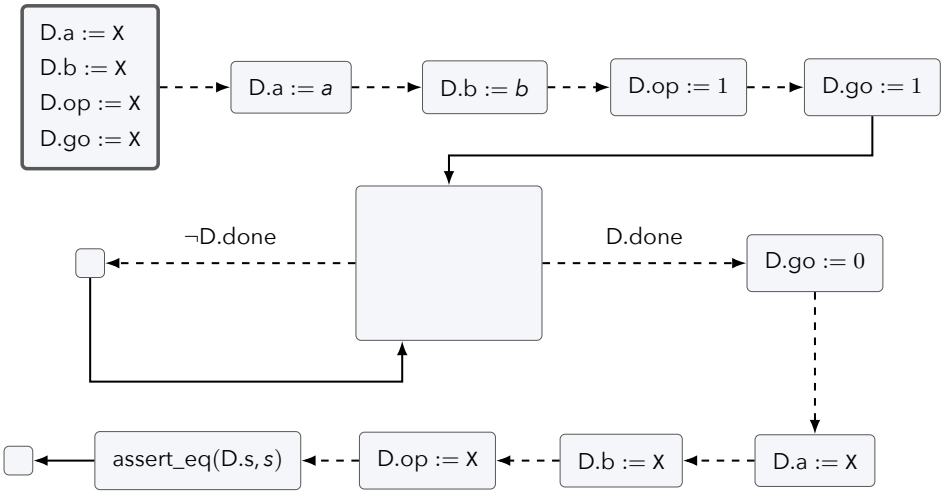



## Top-level lowering

`prot name<D: RTL>(params) { B }`

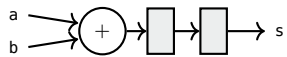


# Unaligned multiplication

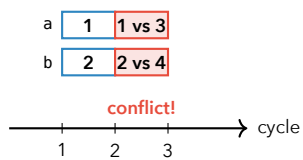


# Recall: `fork()` Conflict without DontCare

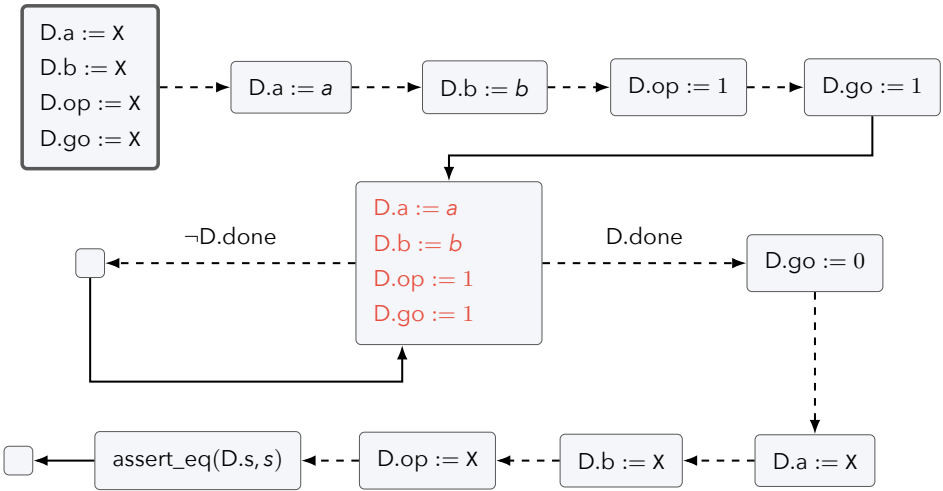
```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  fork();
  step();
  assert_eq(DUT.s, s);
}
```



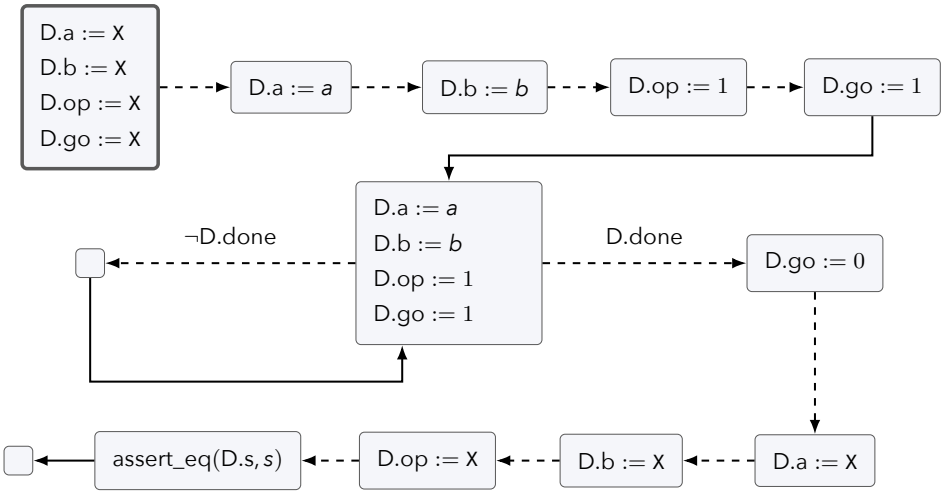

Call: `add(1,2,3); add(3,4,7)`



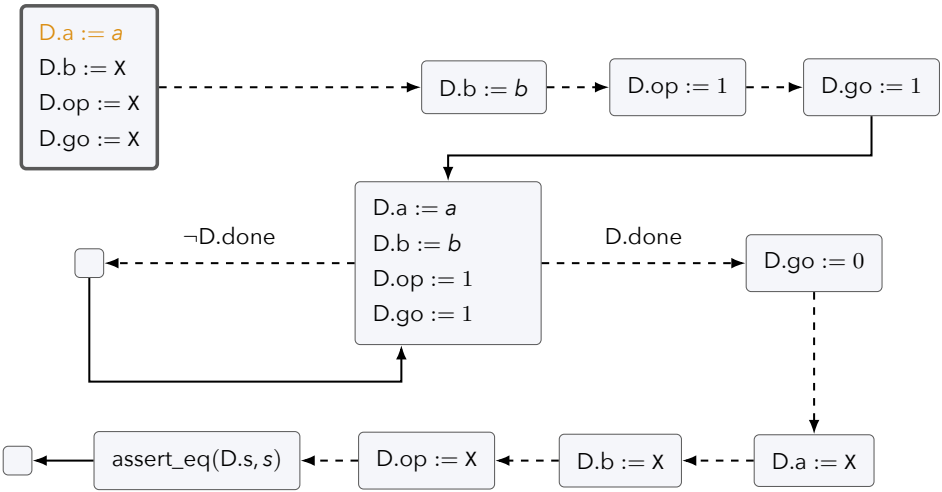
# Persistent assignments



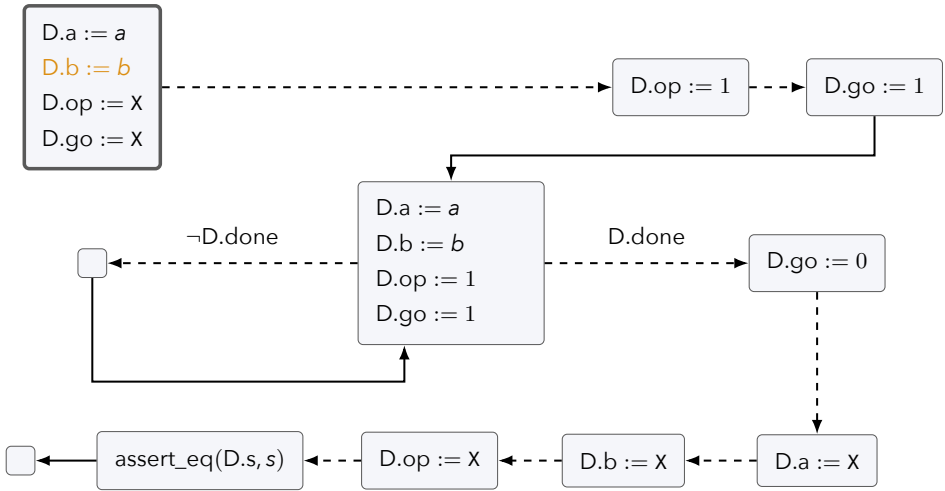
# Clock alignment



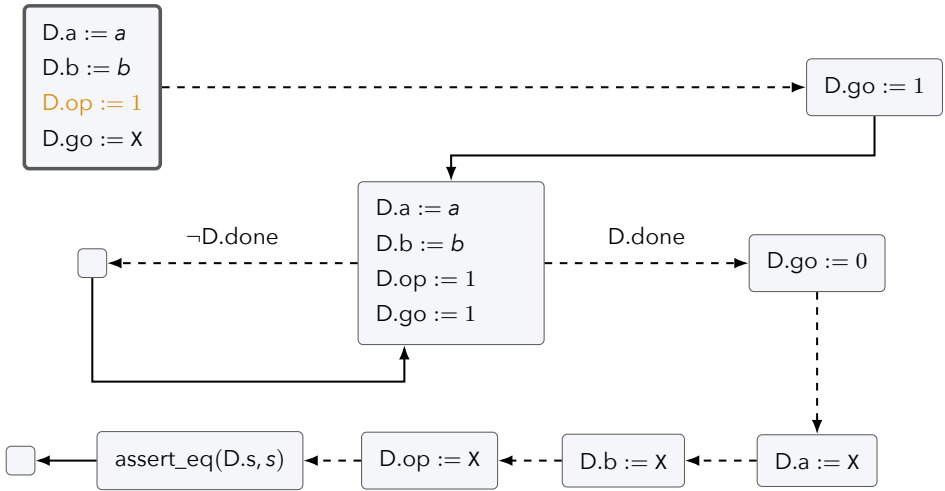
# Clock alignment



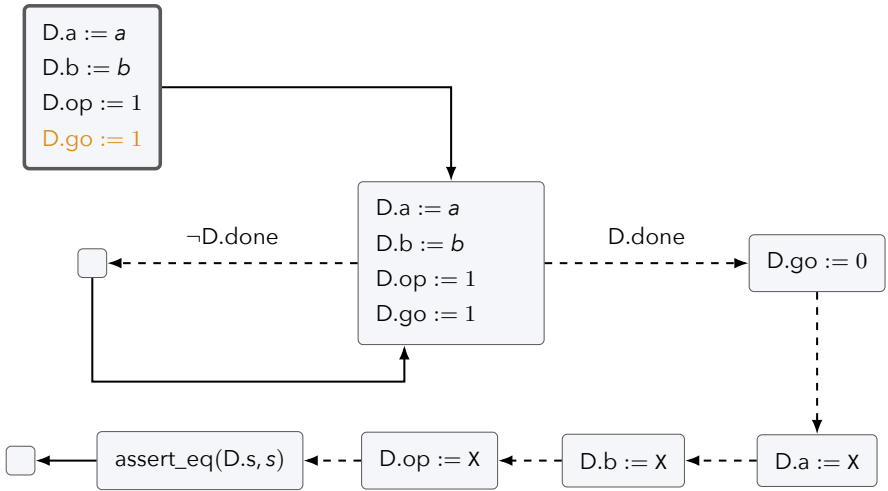
# Clock alignment



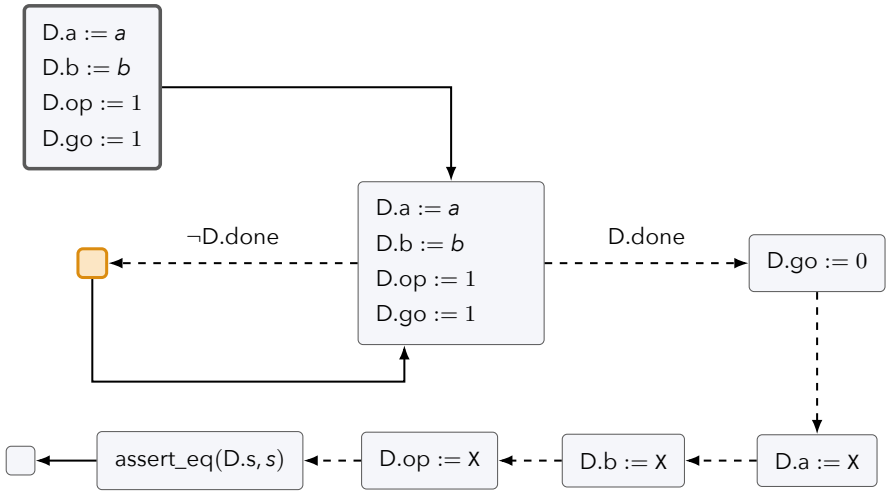
# Clock alignment



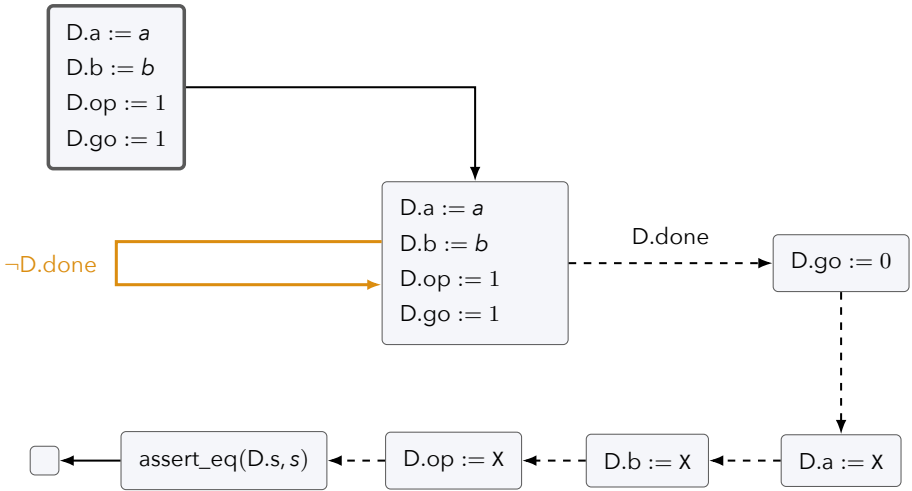
# Clock alignment



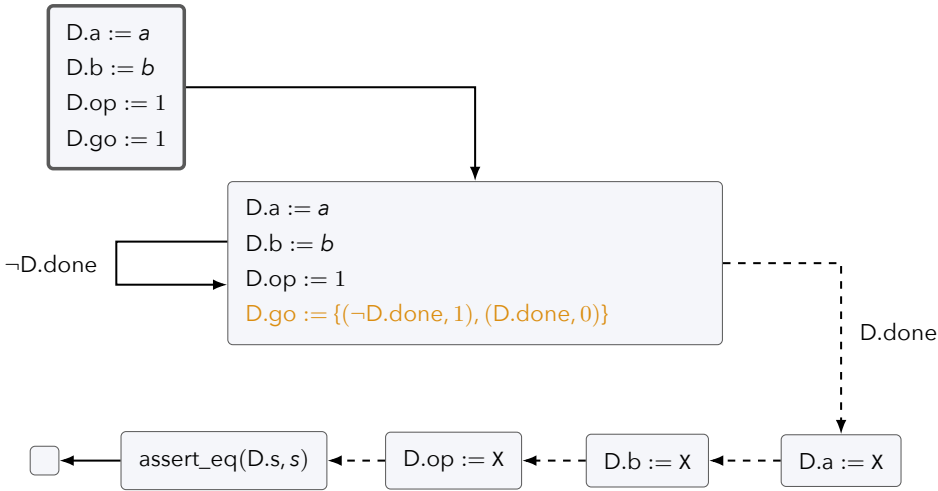
# Clock alignment



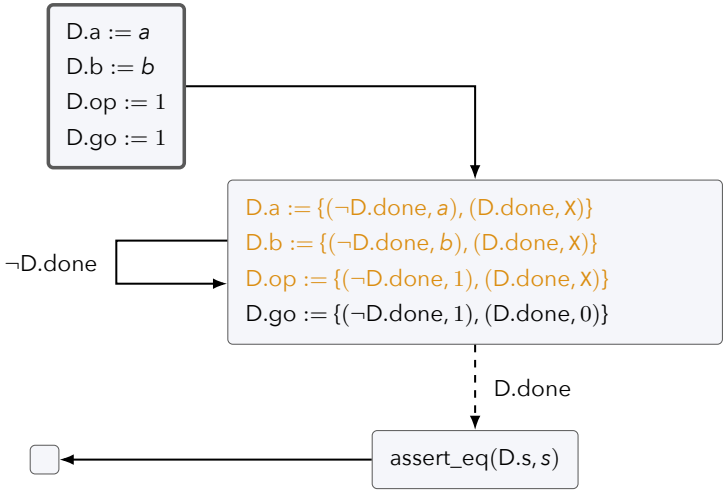
# Clock alignment



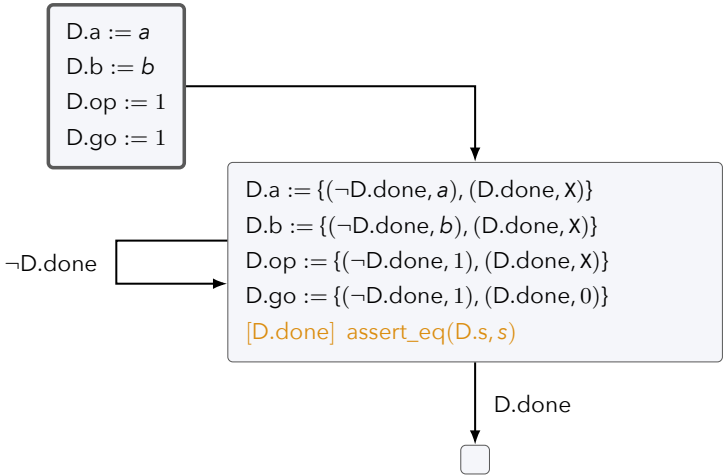
# Clock alignment



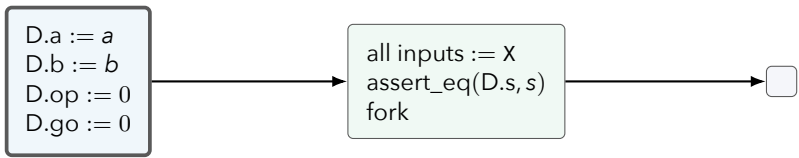
# Clock alignment



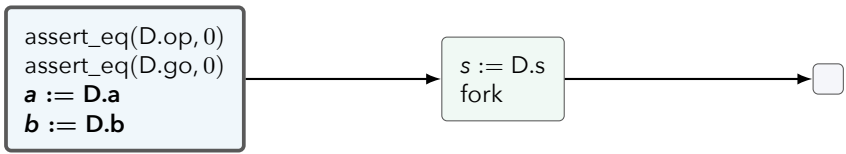
# Clock alignment



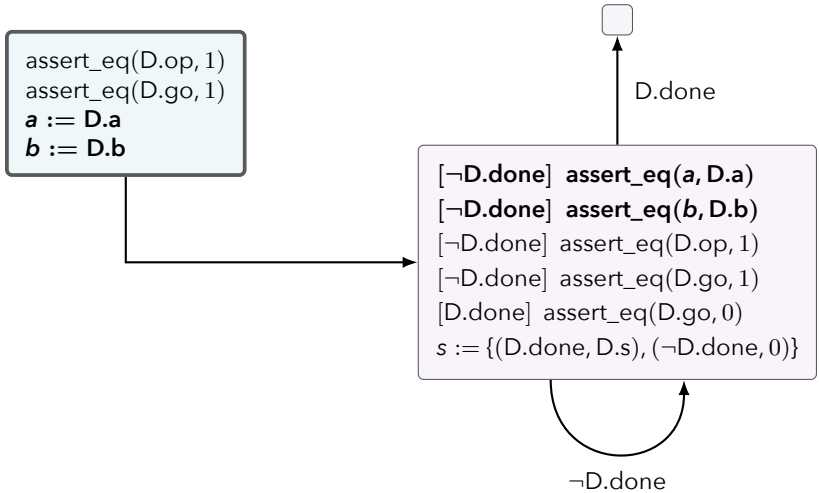
# Clock-aligned addition



# Single-transaction monitors

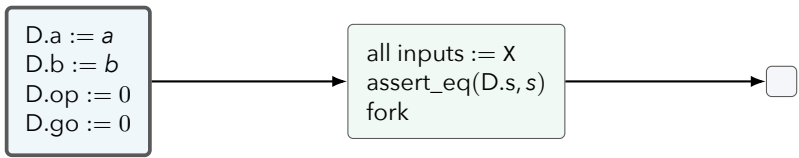


# Single-transaction monitors

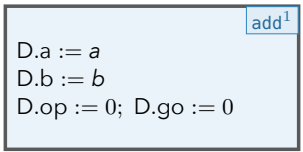


# Multi-Protocol, Multi-Transaction Compilation

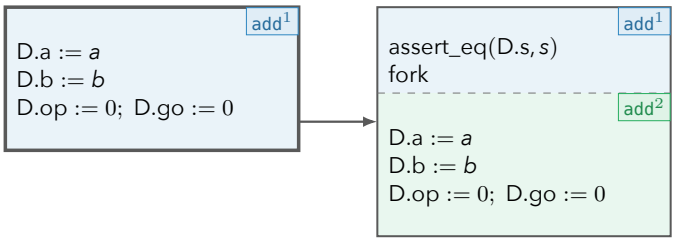
# Clock-aligned addition



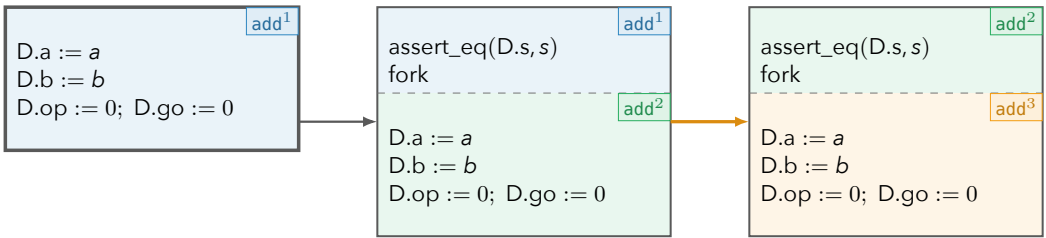
# Building the adder driver



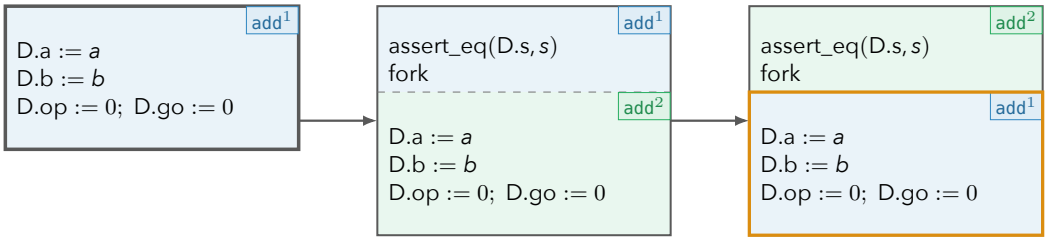
# Building the adder driver



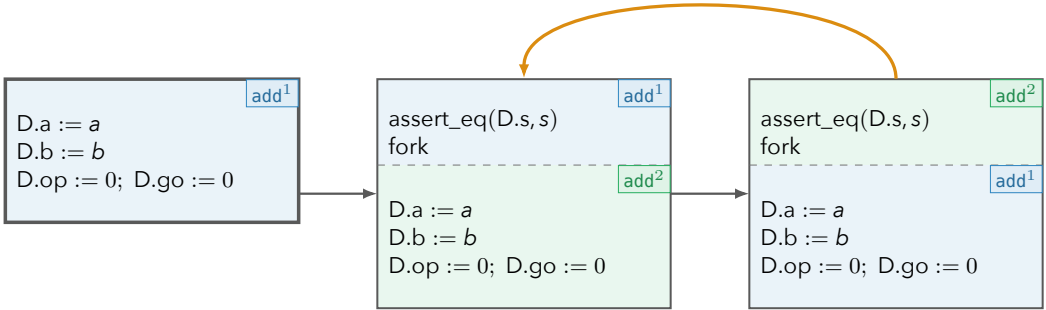
# Building the adder driver



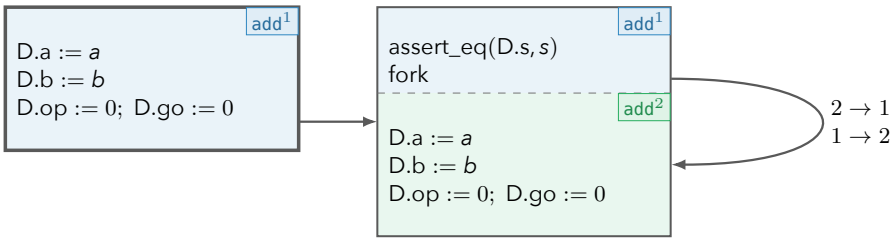
# Building the adder driver



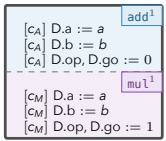
# Building the adder driver



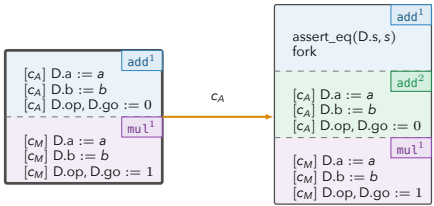
# Building the adder driver



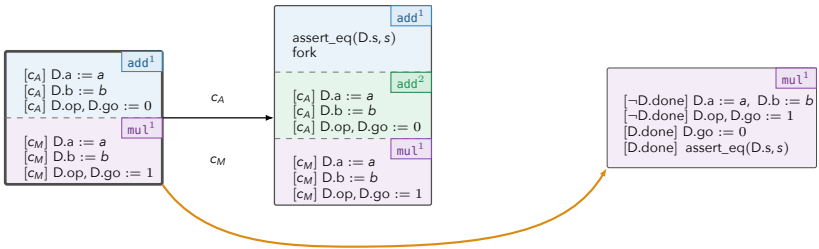
# Building the add/mul driver



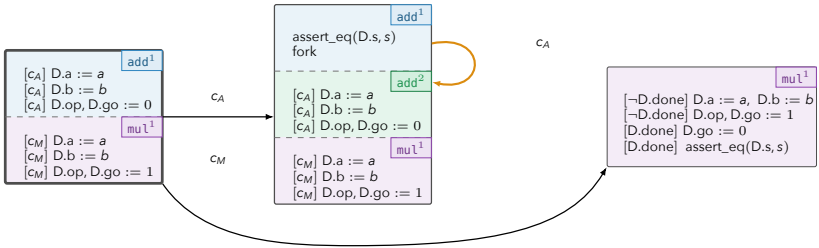
# Building the add/mul driver



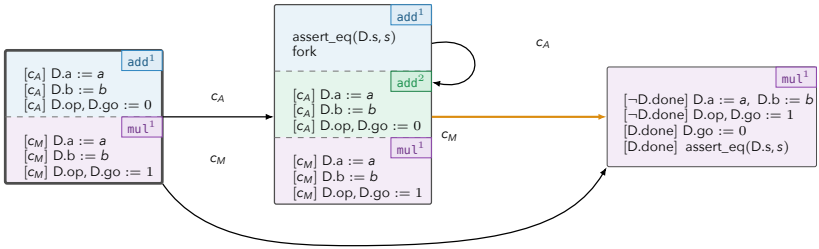
# Building the add/mul driver



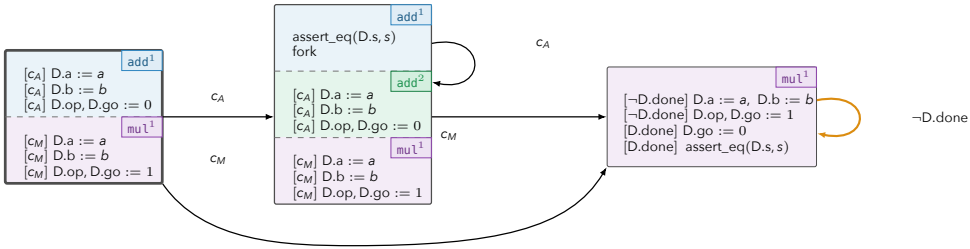
# Building the add/mul driver



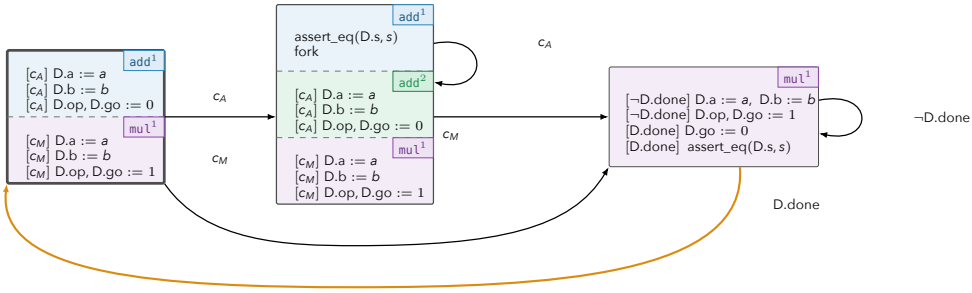
# Building the add/mul driver



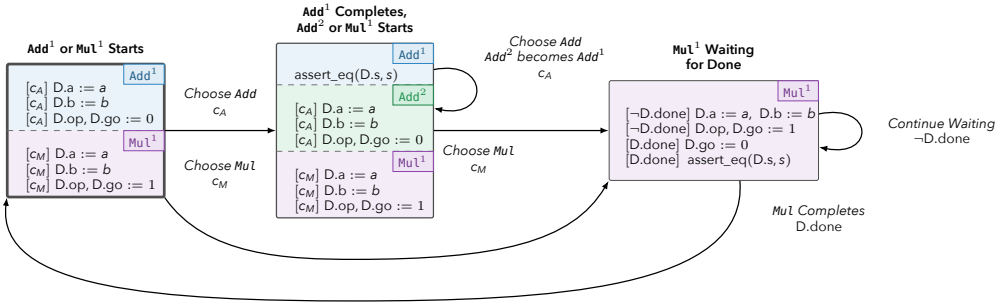
# Building the add/mul driver



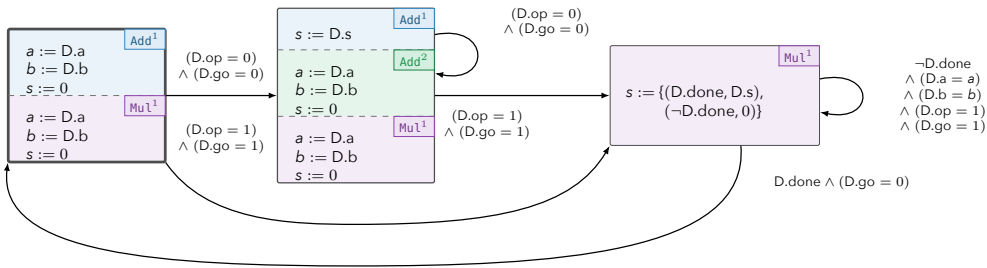
# Building the add/mul driver



# Unbounded Driver



# Unbounded Monitor



# Unbounded waiting

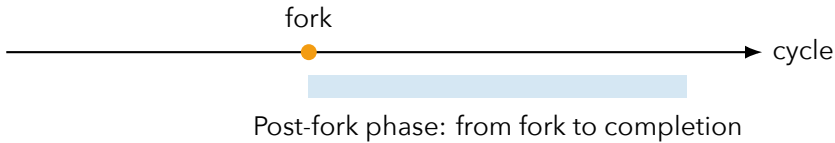
```
prot p<D: RTL>() {
  D.in := 0; step();
  fork(); D.in := X;
  while D.out == 0 { step(); }
  step();
}
```

$$p_1^1 \quad \longrightarrow \quad p_1^1 \wedge p_1^2 \quad \longrightarrow \quad p_1^1 \wedge p_1^2 \wedge p_1^3 \quad \cdots$$

# Post-fork phase

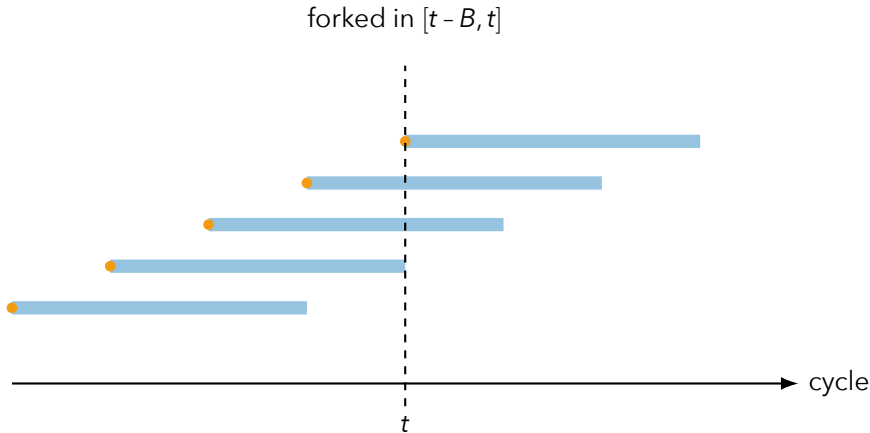


# Post-fork phase




Bounded post-fork phase: completes within $B$ cycles

# Bounded post-fork phase



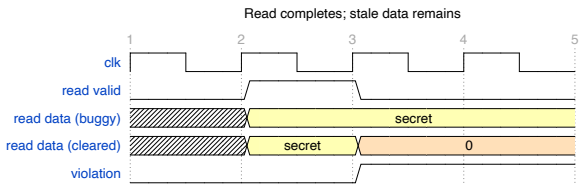

At most $B + 2$ live transactions.

# Applications

# Synthesizable security monitors

- Start from AXI security bugs in eXpect and Xray

## Example: stale read data




Secret remains visible after the read completes

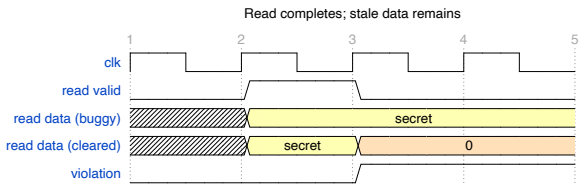
Zonta-Roudes et al., *eXpect: On the Security Implications of Violations in AXI Implementations*, ICCAD 2024

Zonta-Roudes et al., *Xray: Detecting and Exploiting Vulnerabilities in Arm AXI Interconnects*, DATE 2025

# Synthesizable security monitors

- Start from AXI security bugs in eXpect and Xray
- Translate the security properties to Wishbone

## Example: stale read data




Secret remains visible after the read completes

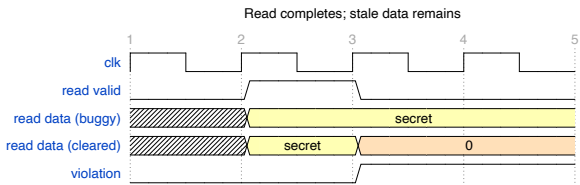

Zonta-Roudes et al., *eXpect: On the Security Implications of Violations in AXI Implementations*, ICCAD 2024

Zonta-Roudes et al., *Xray: Detecting and Exploiting Vulnerabilities in Arm AXI Interconnects*, DATE 2025

# Synthesizable security monitors

- Start from AXI security bugs in eXpect and Xray
- Translate the security properties to Wishbone
- Synthesize a monitor alongside a LiteX SoC

## Example: stale read data




Secret remains visible after the read completes


Zonta-Roudes et al., *eXpect: On the Security Implications of Violations in AXI Implementations*, ICCAD 2024

Zonta-Roudes et al., *Xray: Detecting and Exploiting Vulnerabilities in Arm AXI Interconnects*, DATE 2025

# FFI for Hardware

- HLS compilers expect a small set of predefined interfaces
- Existing RTL components often use different protocols
- Integrating them requires handwritten adapters

# Call legacy RTL from Calyx

```
prot push<D: Fifo>(
  inp: u32
) {
  D.enq := 1'b1;
  D.vld := 1'b1;
  D.inp := inp;
  step();
}
prot pop<D: Fifo>(
  out: u32
) {
  D.enq := 1'b0;
  D.vld := 1'b1;
  step(); fork();
  assert_eq(D.o,out);
  step();
}
```

(a) Paso spec

# Call legacy RTL from Calyx

```
prot push<D: Fifo>(
  inp: u32
) {
  D.enq := 1'b1;
  D.vld := 1'b1;
  D.inp := inp;
  step();
}
prot pop<D: Fifo>(
  out: u32
) {
  D.enq := 1'b0;
  D.vld := 1'b1;
  step(); fork();
  assert_eq(D.o,out);
  step();
}
```

(a) Paso spec

```
repeat 10 { // push 10
      vals
  invoke fifo(op=1'b0,
    inp=count_32.out)();
  incr;
}
invoke count(in=4'd0)();
repeat 10 { // pop 10 vals
  invoke fifo(op=1'b1)();
  invoke use(in=fifo.out)
      ();
  incr;
}
```

(b) Calyx HLS code

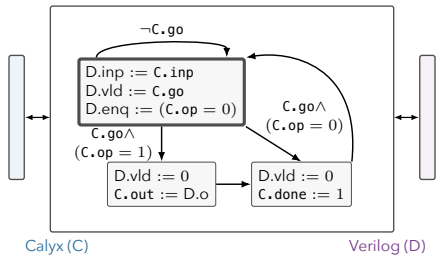
# Call legacy RTL from Calyx

```
prot push<D: Fifo>(
  inp: u32
) {
  D.enq := 1'b1;
  D.vld := 1'b1;
  D.inp := inp;
  step();
}
prot pop<D: Fifo>(
  out: u32
) {
  D.enq := 1'b0;
  D.vld := 1'b1;
  step(); fork();
  assert_eq(D.o,out);
  step();
}
```

(a) Paso spec

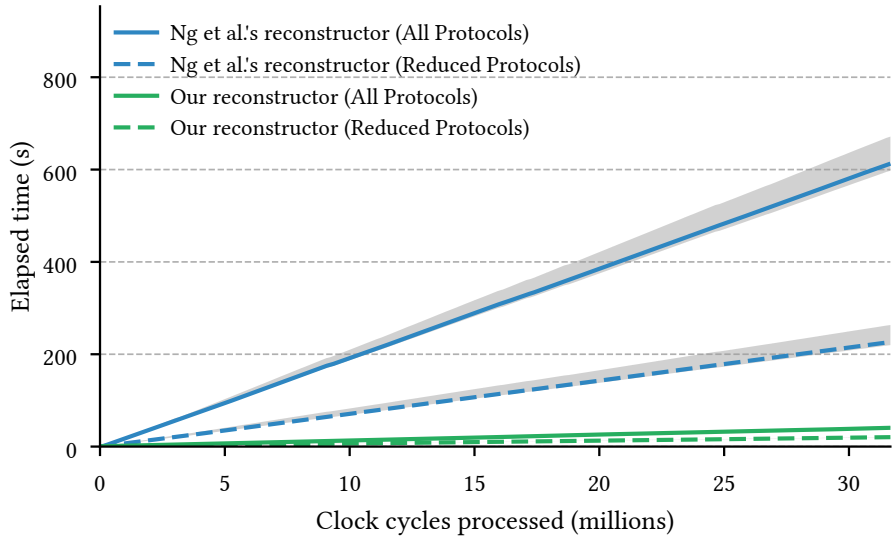
```
repeat 10 { // push 10
      vals
  invoke fifo(op=1'b0,
    inp=count_32.out)();
  incr;
}
invoke count(in=4'd0)();
repeat 10 { // pop 10 vals
  invoke fifo(op=1'b1)();
  invoke use(in=fifo.out)
      ();
  incr;
}
```

(b) Calyx HLS code




(c) Generated adapter

# Faster software reconstruction

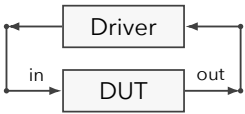


# Future work

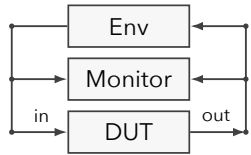
Beyond drivers and monitors

# Three useful interpretations

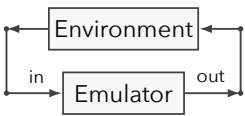
**Driver**




**Monitor**

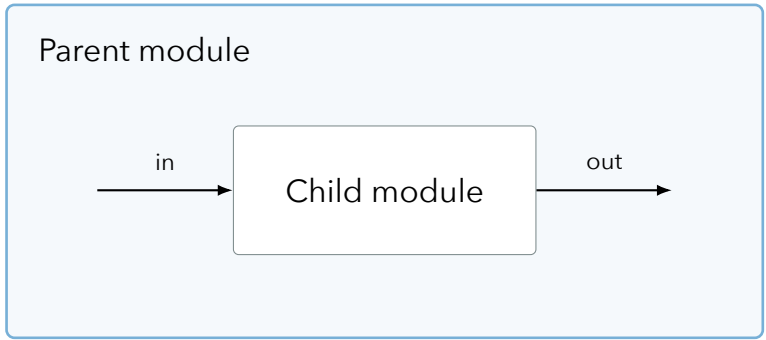



**Emulator**

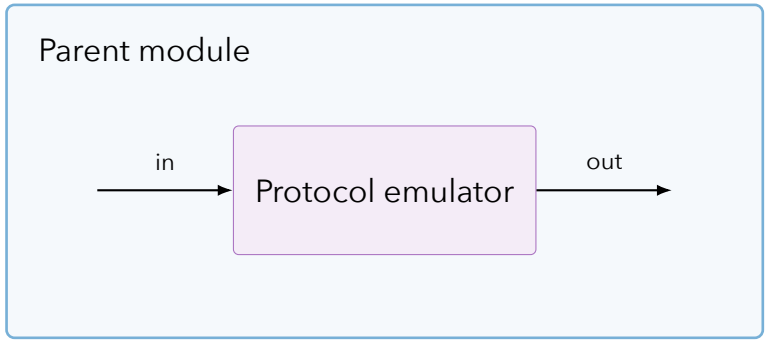



| | Inputs | Outputs |
|---|---|---|
| Driver | drive | observe |
| Monitor | observe | observe |
| Emulator | observe | drive |
| No useful case | drive | drive |

# Emulators enable assume-guarantee reasoning



# Emulators enable assume-guarantee reasoning



# Hierarchical protocols

**Ready-Valid Channel** $\Longrightarrow$ **AXI Write** $\Longrightarrow$ **Memory Subsystem**

```
prot send<C: RV>(x: C.T, n: uint)
{
  C.data := x;
  C.valid := 1;
  // n cycles of  backpressure
  repeat n iterations {
    assert_eq(C.ready, 0);
    step();
  }
  assert_eq(C.ready, 1);
  step();
}
```

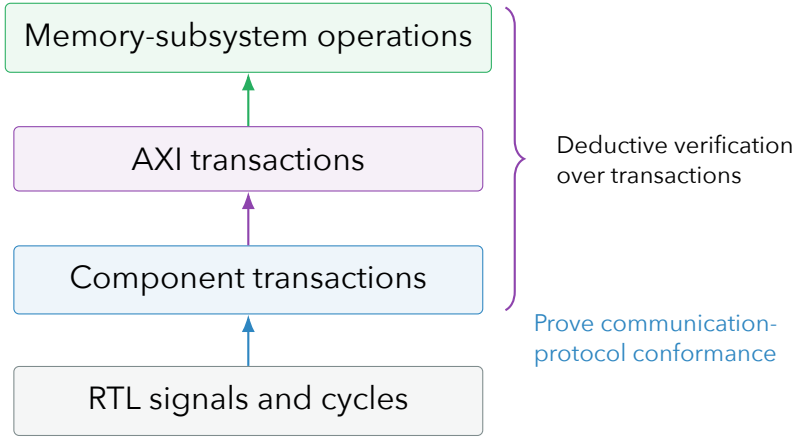
```
composed write<A: AXI>(
  id: u8, addr: u64, data: u64,
  n1: uint, n2: uint) {
  par { // concurrent channels
    A.AW.send(addr, n1);
  } and {
    A.W.send(data, n2);
  }
  A.B.recv(id, resp);
  assert_eq(resp, OKAY);
}
```

```
system store<M: Memory>(
  core: u8, va: u64, x: u64,
  n1: uint, n2: uint) {
  pa := M.tlb.translate(core, va);
  M.coherence.acquireM(core, pa);

  M.l1.write(core, pa, x);
  id := M.ids.fresh(); // write-
through

  M.axi.write(id, pa, x, n1, n2);
}
```

# Deductive verification



# Property-based testing over transactions

```
let trace1 match:
  add(b, c, temp1)
  add(a, temp1, s1)
let trace2 match:
  add(a, b, temp2)
  add(temp2, c, s2)
assert:
  s1 == s2
```

# A round trip through a FIFO

```
let trace1 match:
  push(x)
  pop(y)
where:
  x == y
```

# Conclusion

Write **one, formal protocol specification,**

many correct-by-construction uses



Nikil Shyamsunder



Ernest Ng



Francis Pham



Adrian Sampson



Kevin Laeufer

**Code**

github.com/cucapra/protocols

**Paso paper**

arxiv.org/pdf/2606.13659

**Home Page**

nikilshyamsunder.com