Capra lab @ Cornell

Paso: Specifying RTL Communication as Programs

Ernest Ng, Nikil Shyamsunder, Francis Pham, Adrian Sampson, Kevin Laeufer

Cornell CSL Seminar
September 25, 2026

Calling Conventions in Software

Standardized contract about how to invoke functions

Calling Conventions in Software

Given a calling convention,
there is exactly one way to invoke a function

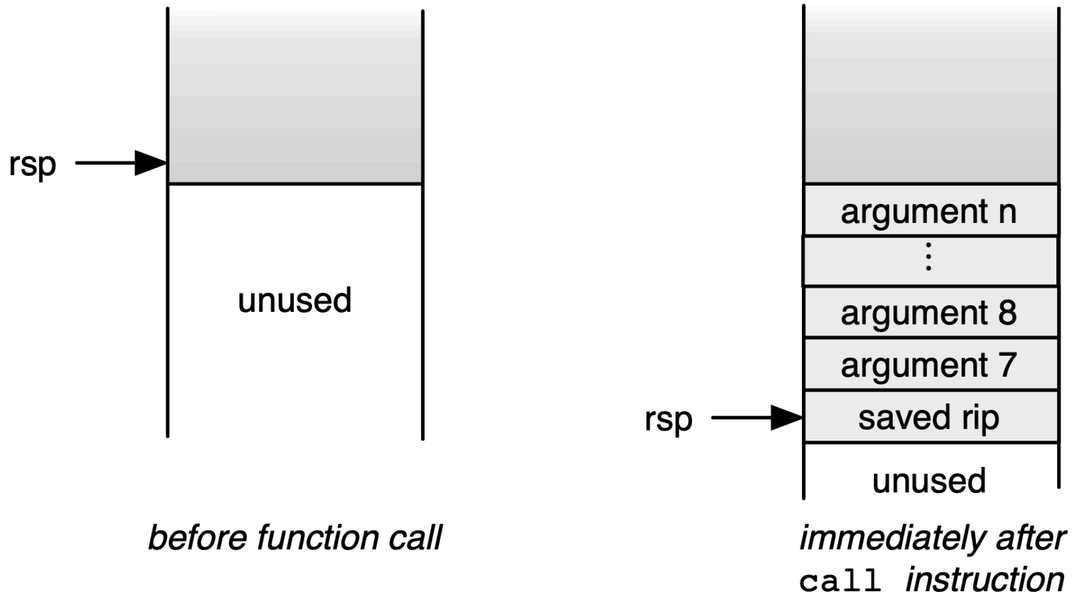


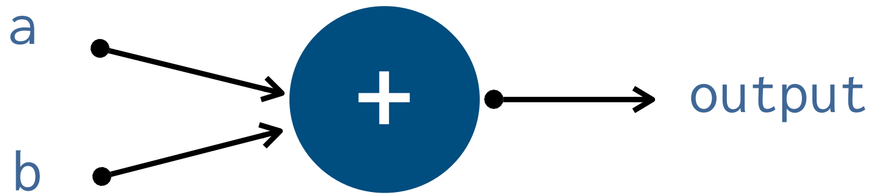
Figure from Andrew Myers

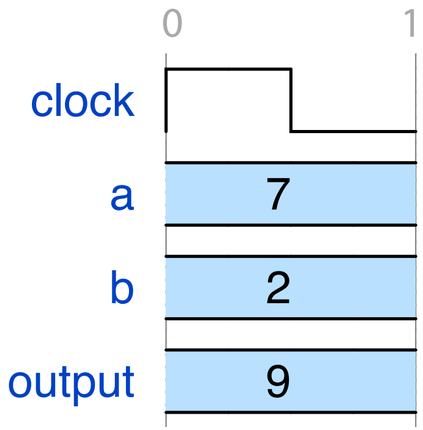
Calling Conventions in *Hardware*?

There is no one way to "call" a HW module!

“Calling” an adder in hardware

Combinational Adder



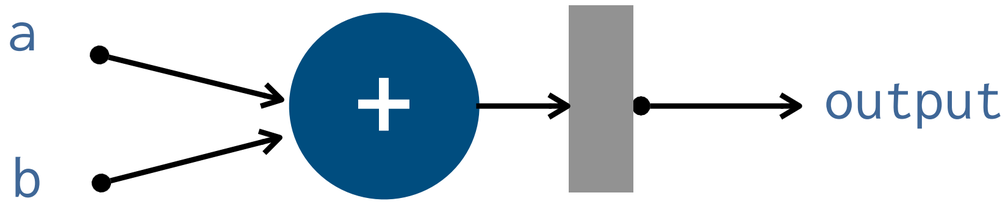


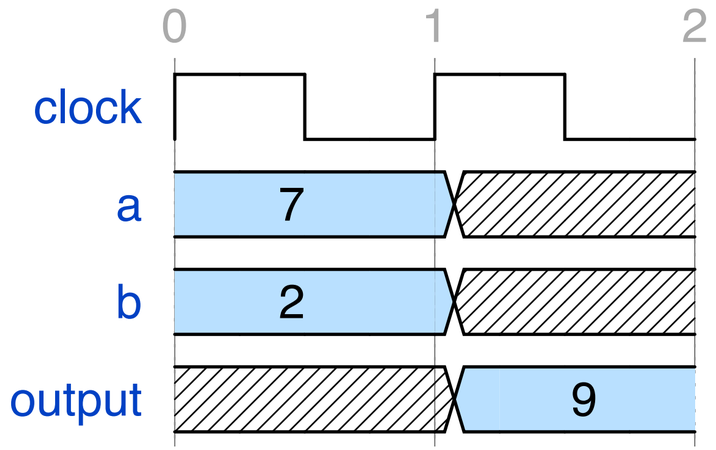
“If I drive `a = 7`, `b = 2`,

I get `output = 9` in the ***same*** clock cycle”

“Calling” an adder in hardware

Sequential Adder





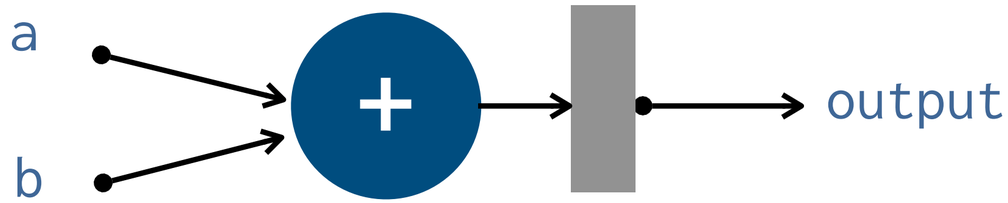
“If I drive `a = 7`, `b = 2`,

I get `output = 9` ***after*** one clock cycle”

“Calling” a hardware module depends on how the module is implemented!

The abstraction gap in hardware

Hardware runs at the level of individual clock cycles & signals,
but we care about high-level **transactions**!



Transactions

"Adding 7 and 2
results in 9"

Signals

"If I drive `a = 7`, `b = 2`,
I get `output = 9` after one clock cycle"

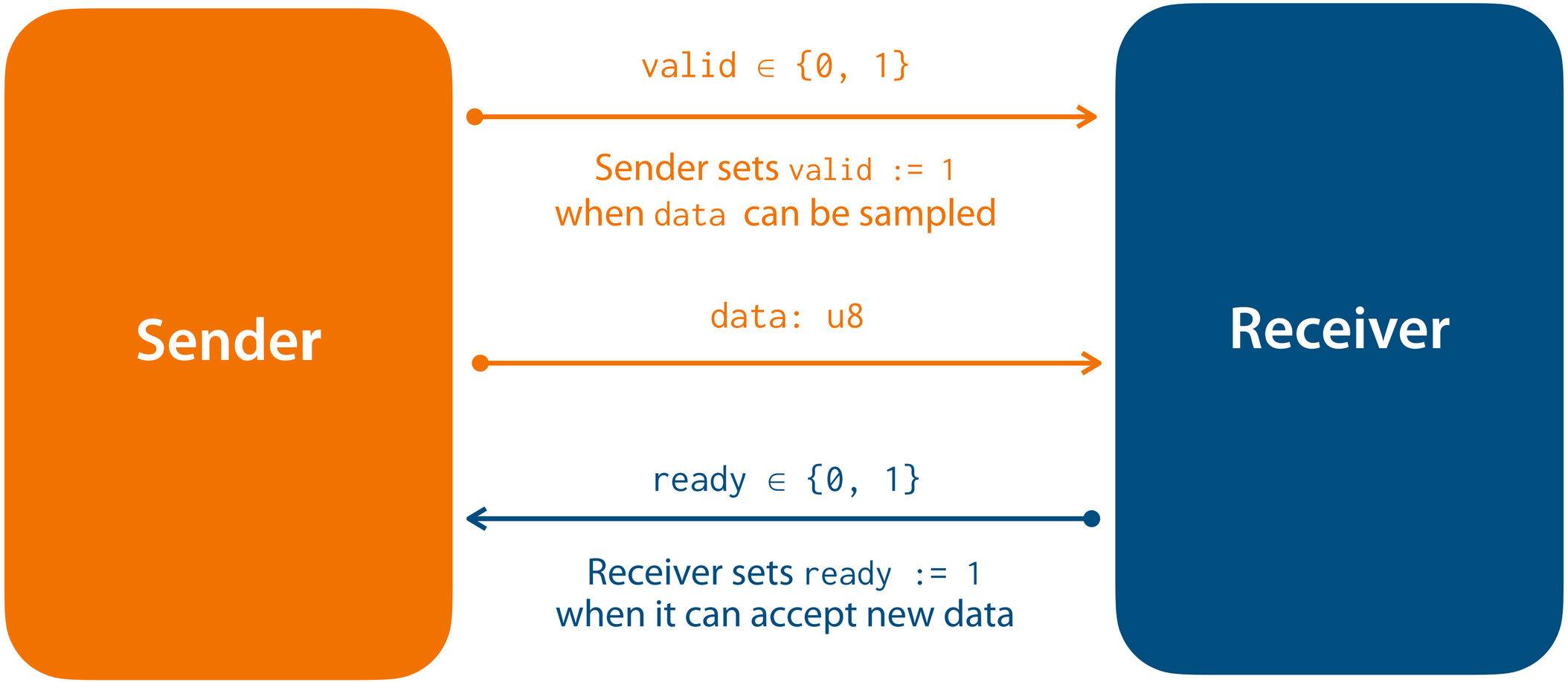
"Calling" a hardware module
involves **communication protocols**

(I/O signals changing
over clock cycles)

Example Protocol: Ready-Valid Handshake

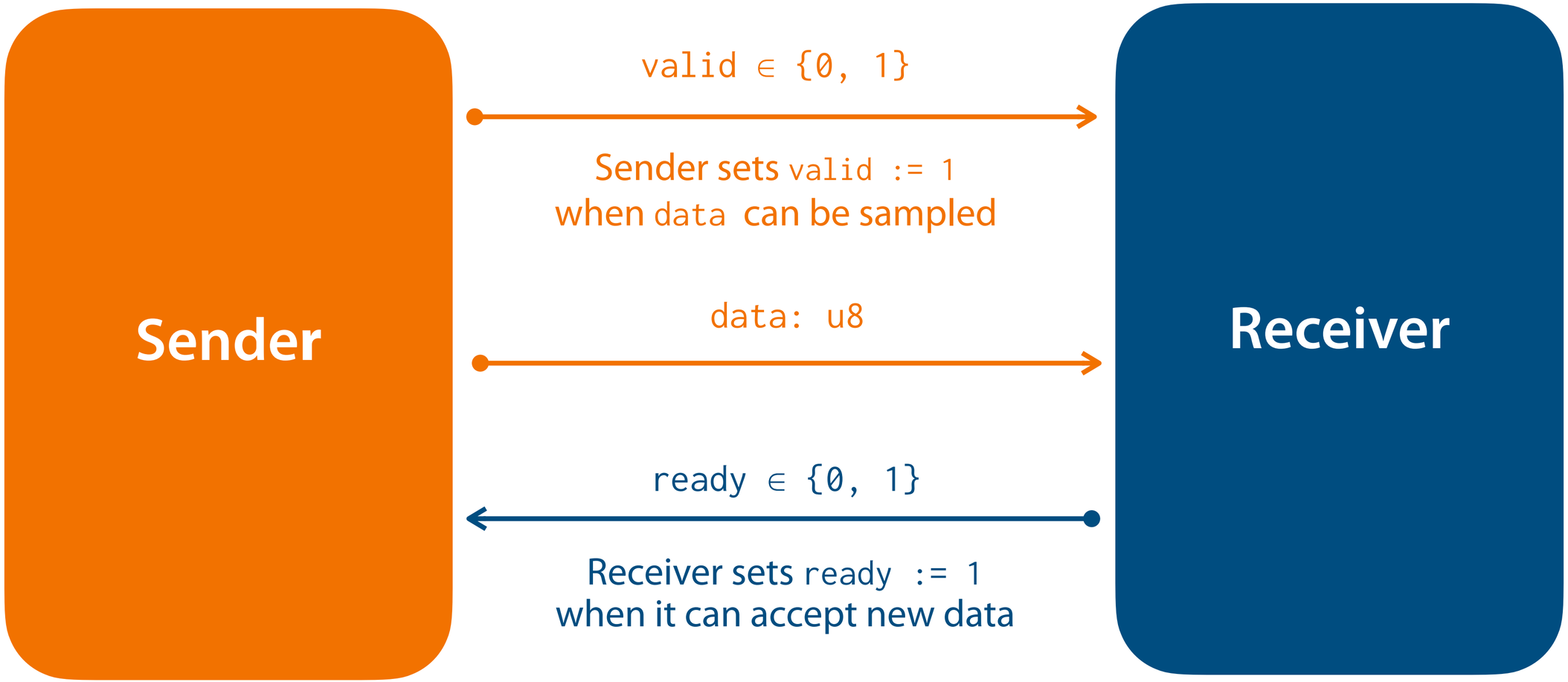
Ready-Valid Handshake

Mediates data transfer between a sender & a receiver



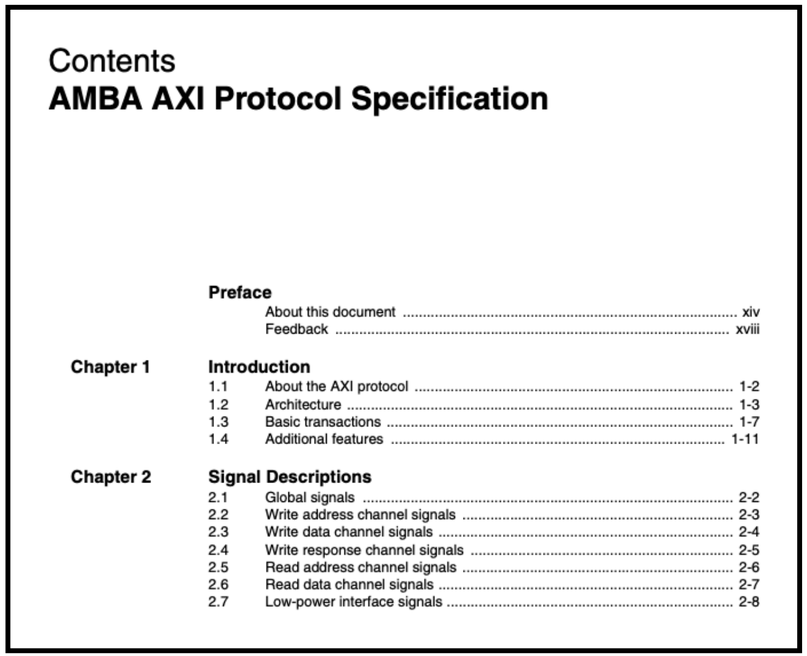
Ready-Valid Handshake

Invariant: `data` is only sent when `ready` & `valid` are both 1 during the same clock cycle



Problem: Protocols are often stated informally / implicitly

Scattered across specification documents, comments, assertions, . . .



Contents
AMBA AXI Protocol Specification

AXI specification from ARM

Susceptible to communication bugs!

[Ma et al. ASPLOS '22]

Debugging in the Brave New World of Reconfigurable Hardware

Jiacheng Ma
University of Michigan

Gefei Zuo
University of Michigan

Kevin Loughlin
University of Michigan

Haoyang Zhang
University of Michigan

Andrew Quinn
University of California, Santa Cruz

Baris Kasikci
University of Michigan

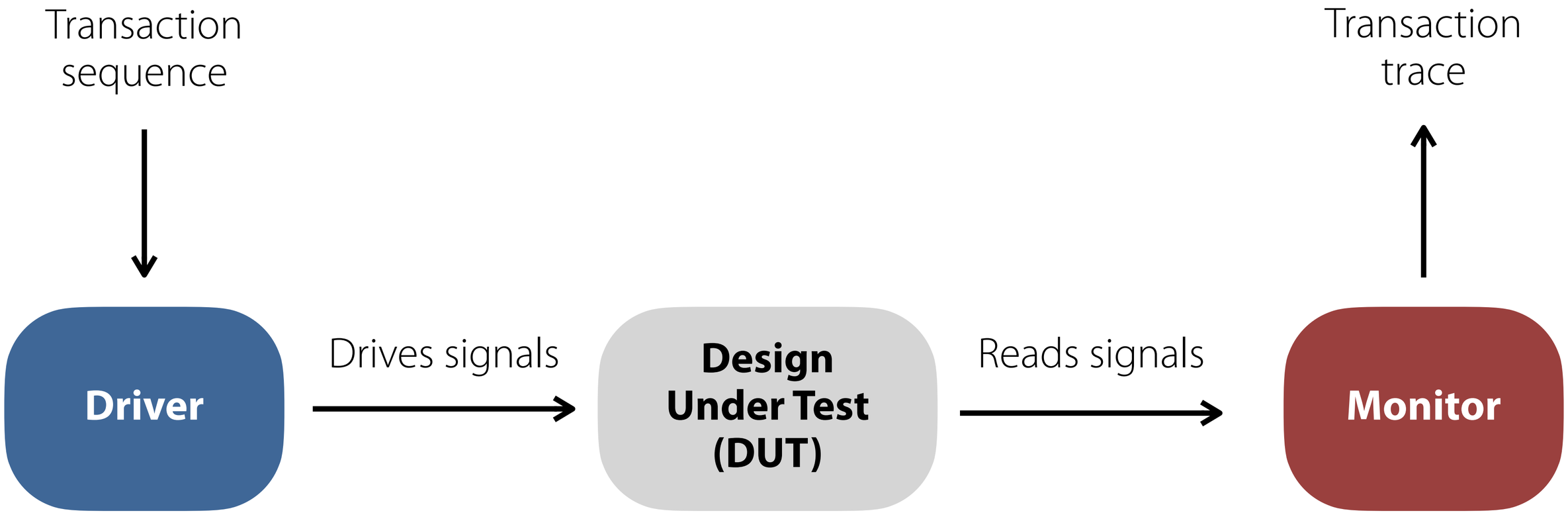
Problem: Protocols are often stated informally / implicitly

We're interested in both:

- Standardized protocols (e.g., AXI-Stream, Wishbone)

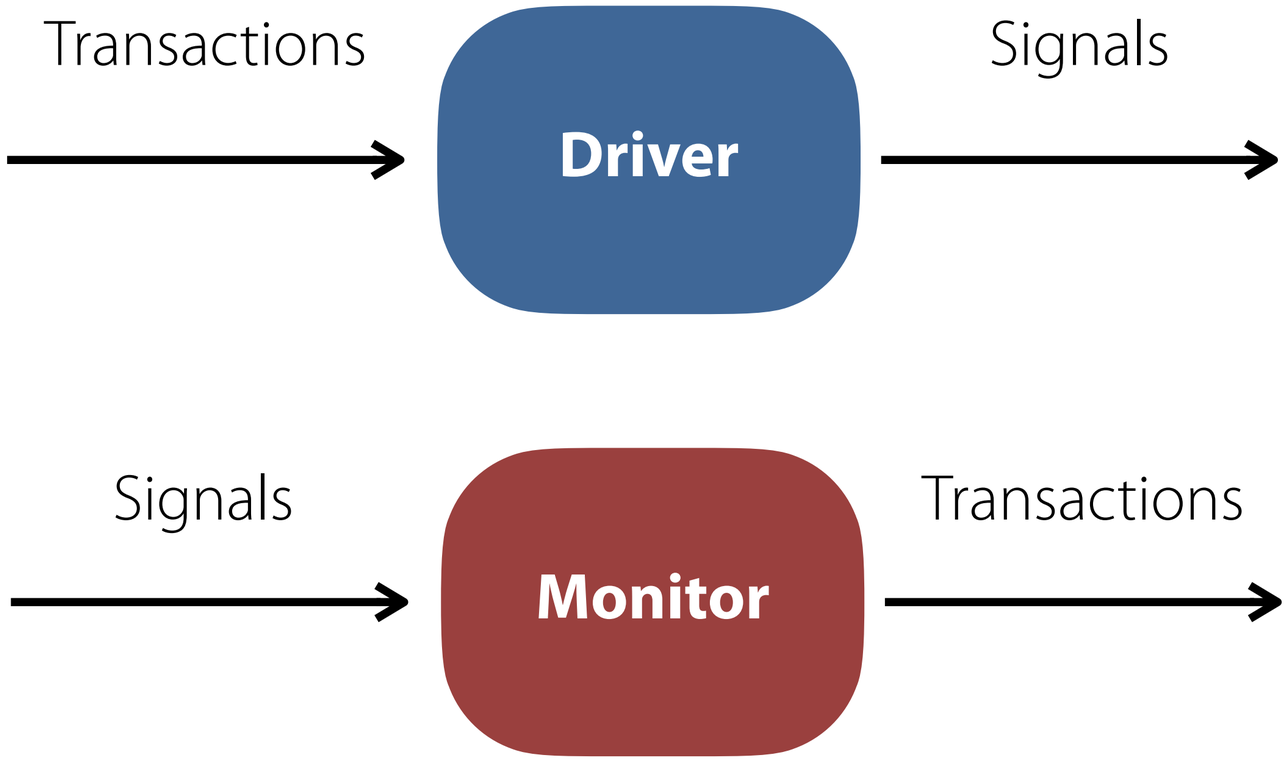
- Simple handshakes (e.g., Ready-Valid) & functional units (e.g., how a processor pipeline invokes its ALU)

Typical hardware testing & debugging workflow

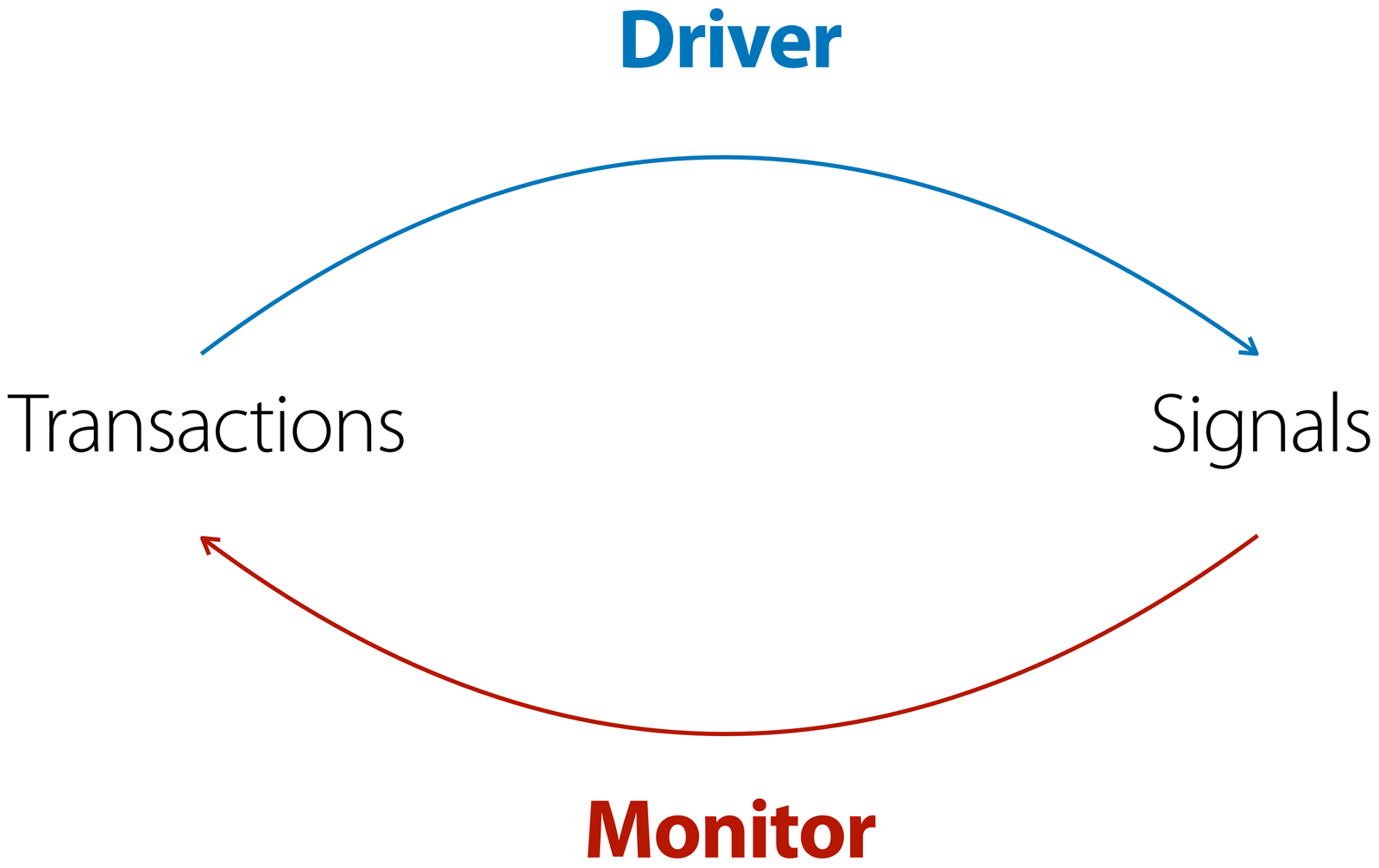


Drivers & Monitors are typically implemented separately

$\Longrightarrow$ inconsistencies!



Observation: Drivers & Monitors are mirror images!



What if we could write just one single program for the driver & monitor?

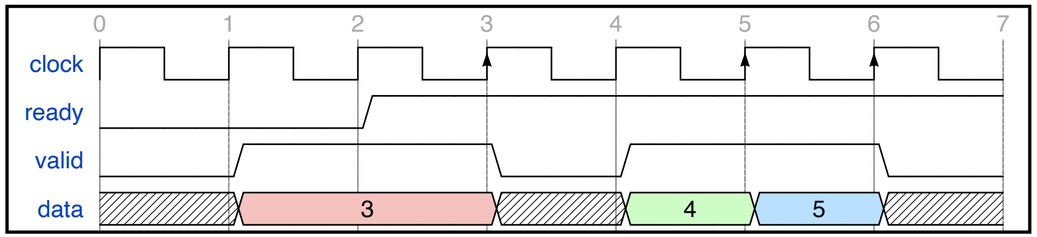
Paso: A DSL for specifying HW communication protocols

```
prot send_data<DUT: ReadyValid>(data: u8) { … }
```

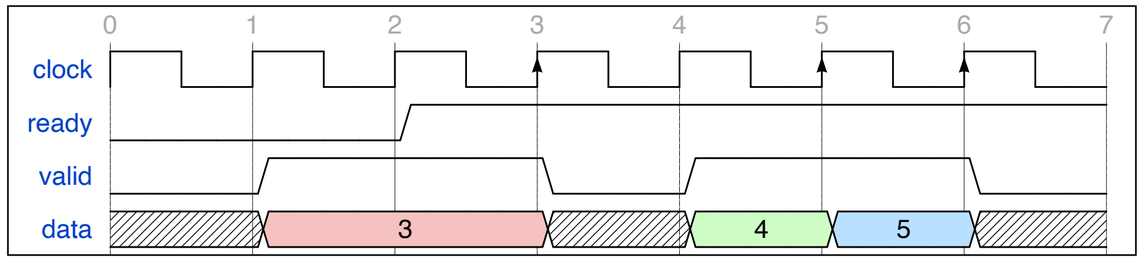
Simple imperative semantics, just like software!

What can we do with one single protocol?

1. Run concrete tests (drive hardware modules)



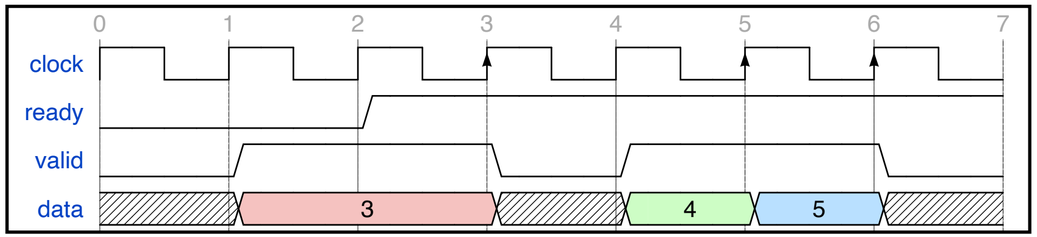
2. Infer transactions from waveforms



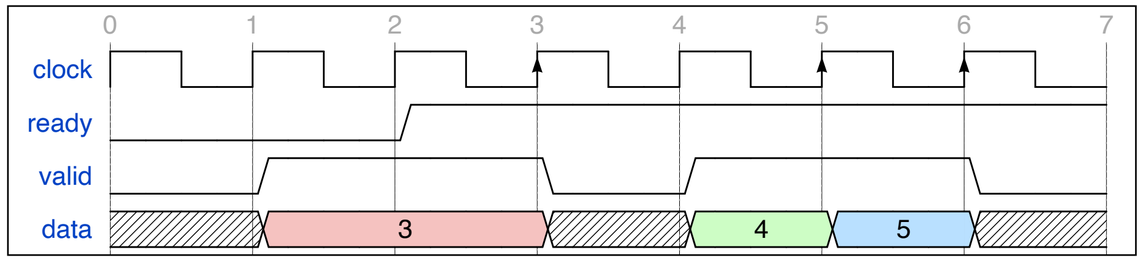

```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

Paso: write one protocol spec, do both!

1. Run concrete tests (drive hardware modules)



2. Infer transactions from waveforms




```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

Paso comes with two tools:

- An **interpreter** (runs tests, like drivers)
- A ***reconstructor*** (infers transactions from signals)

Interpreter

Transactions to execute

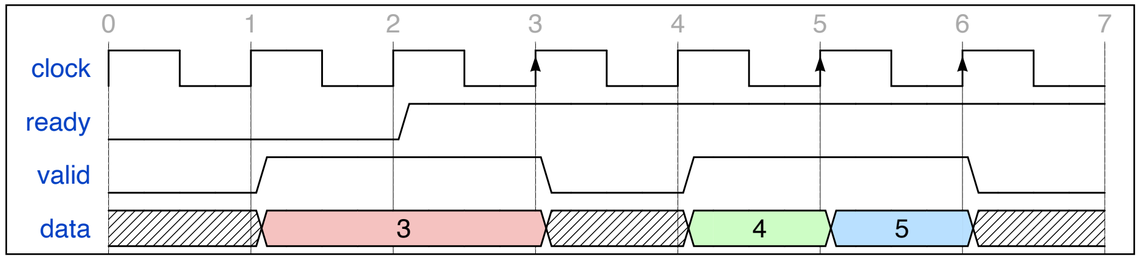
Paso protocol spec

DUT implementation (Verilog)

```
send_data(3);
send_data(4);
send_data(5);
```

Interpreter

Waveform

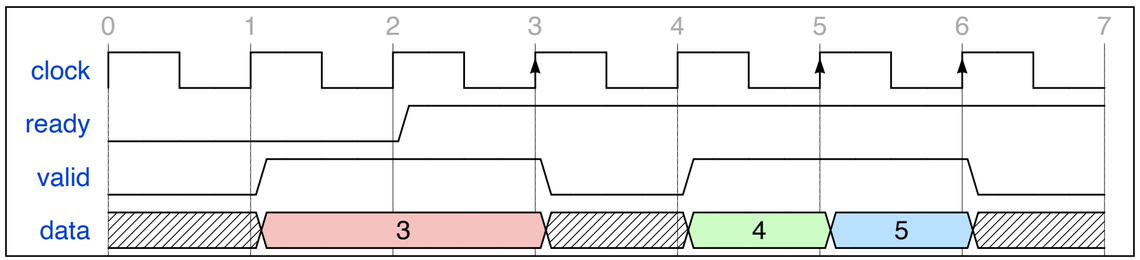


Reconstructor

(the cool part!)

Paso protocol spec

Waveform



Reconstructor

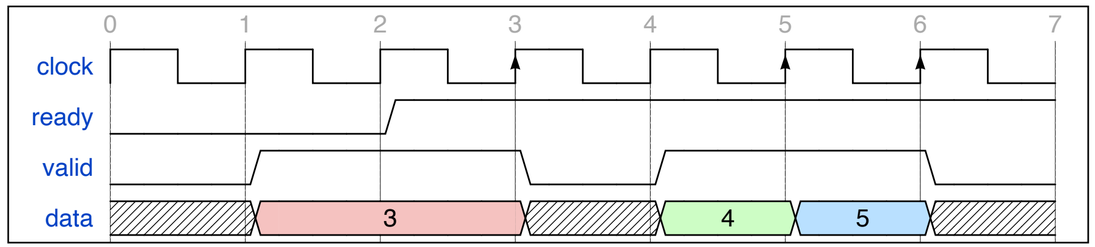
Transaction trace

```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

Applying the Reconstructor on the Ready-Valid Example

Given the protocol definition and a waveform,
the reconstructor infers a series of transactions that are consistent with the waveform

```
prot send_data<DUT: ReadyValid>(data: u8) { … }
```



Reconstructor

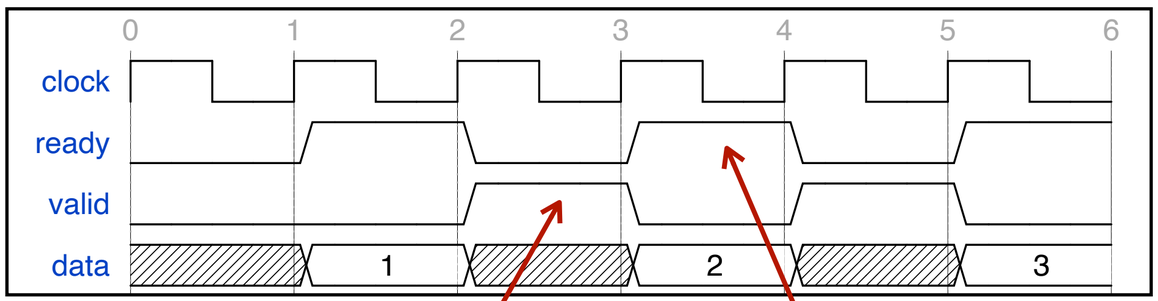
```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

Abstraction level raised from signals to transactions!

Applying the Reconstructor on the Ready-Valid Example

If no transactions could have resulted in the waveform trace,
the reconstructor warns the user

```
prot send_data<DUT: ReadyValid>(data: u8) { … }
```



Reconstructor

```
Error: no matching
transactions!
```

`ready` & `valid` are never both 1
during the same cycle!
(i.e. data transfer never occurs)

Thesis

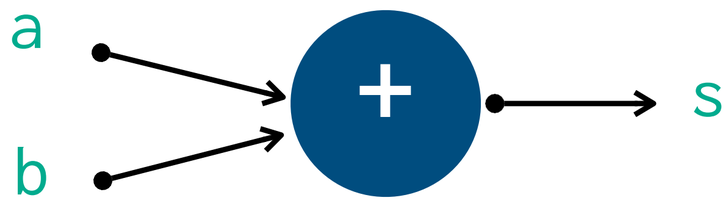
1. Hardware communication protocols can be specified as **programs**! We design a DSL, Paso, which enables this.

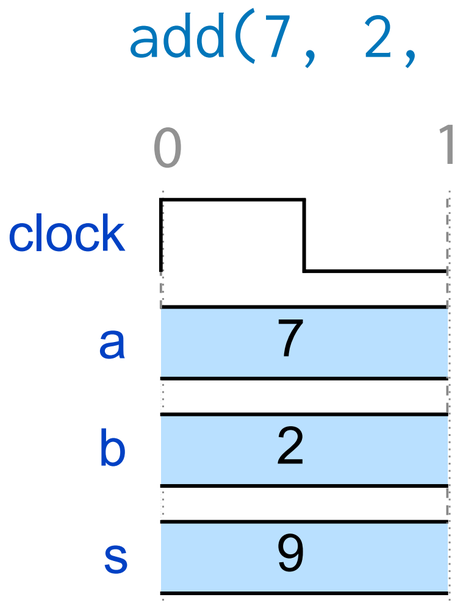
2. Using Paso, we can build PL tools that address pain-points with testing & debugging hardware.

Paso by example:
Two protocol specs for an adder

A combinational adder

Note: we're not checking functional correctness,
we are only interested in the adder's communication interface!



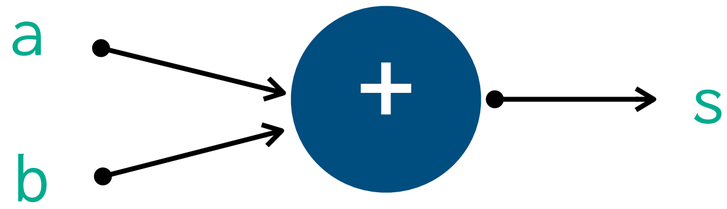


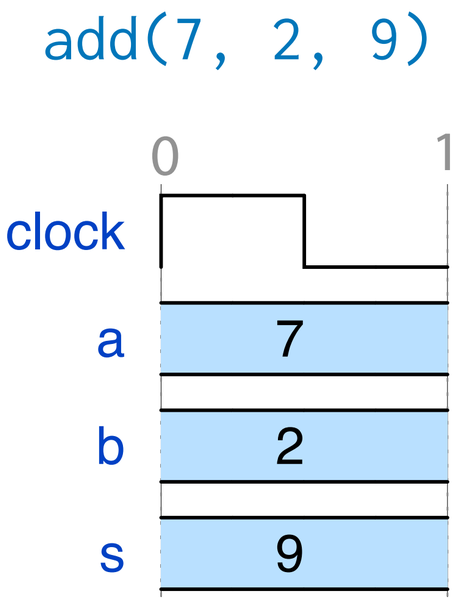
Notation: We specify
expected inputs & outputs
to the transaction,
akin to a unit test

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

A combinational adder

Note: we're not checking functional correctness,
we are only interested in the adder's communication interface!



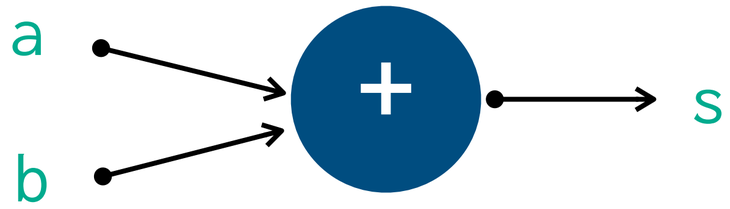


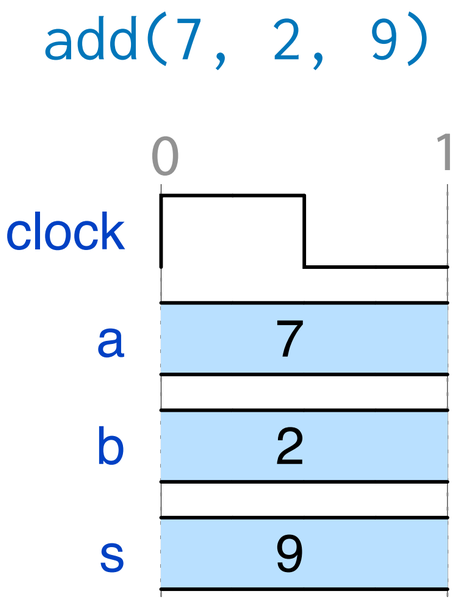

```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

Apply values onto input ports
`DUT.a` & `DUT.b`

A combinational adder

Note: we're not checking functional correctness,
we are only interested in the adder's communication interface!



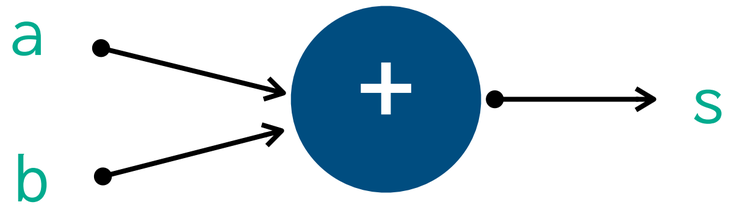


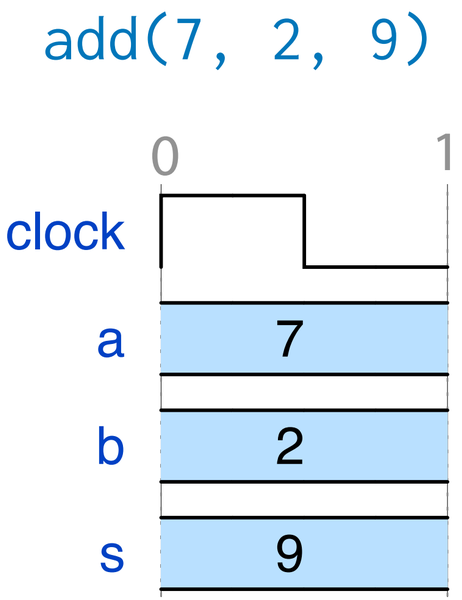

```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

Check that output port `DUT.s`
contains the value `s`

A combinational adder

Note: we're not checking functional correctness,
we are only interested in the adder's communication interface!



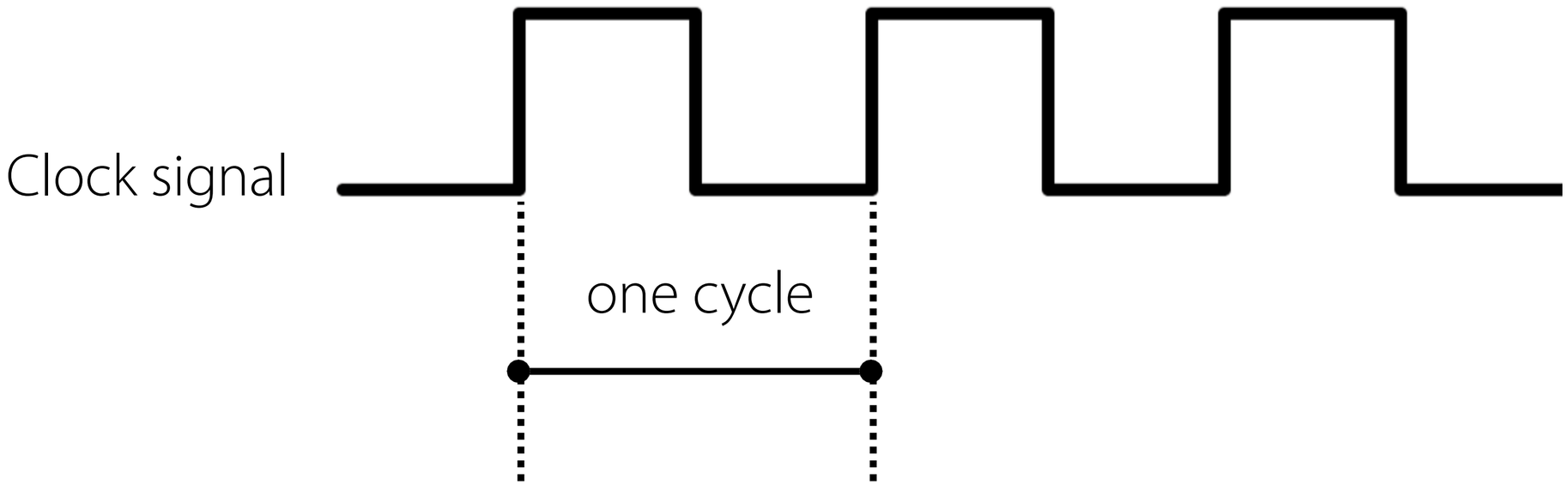



```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

Finish clock cycle

Aside: the `step()` primitive

`step()` advances the clock by one cycle

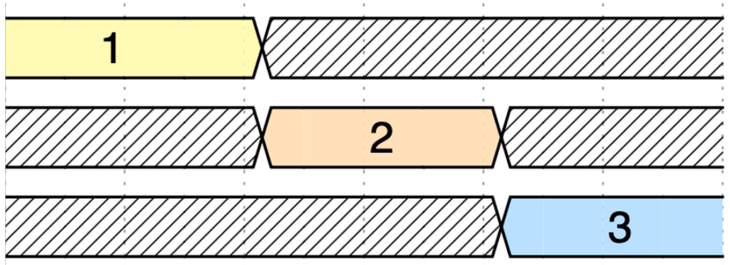


Can we optimize our adder?

Yes, through **pipelining**!

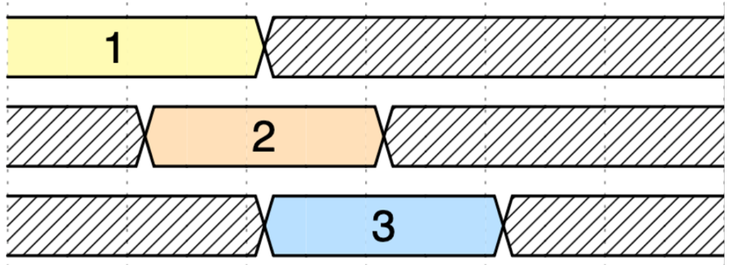
Sequential

Processes **one** input at a time



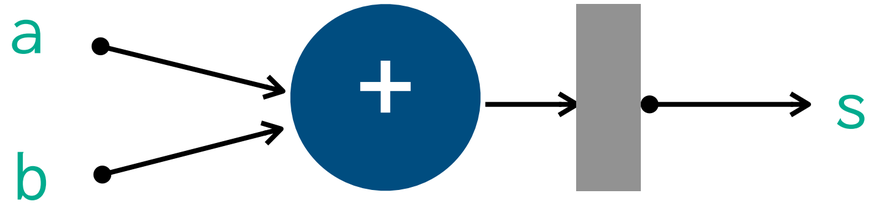
Pipelined

Processes **multiple** inputs at a time



`fork()` allows for concurrent protocol execution in our DSL

A naïve attempt at a pipelined adder



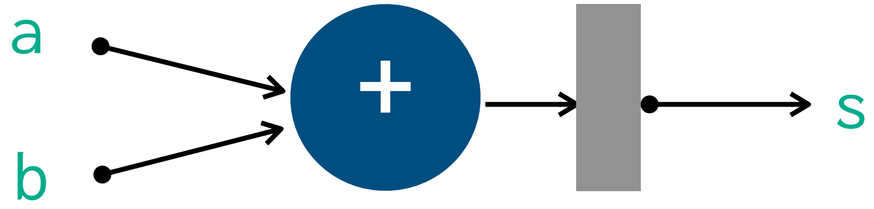
Combinational

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



  assert_eq(DUT.s, s);
  step();
}
```

Pipelined (naïve)

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

A naïve attempt at a pipelined adder



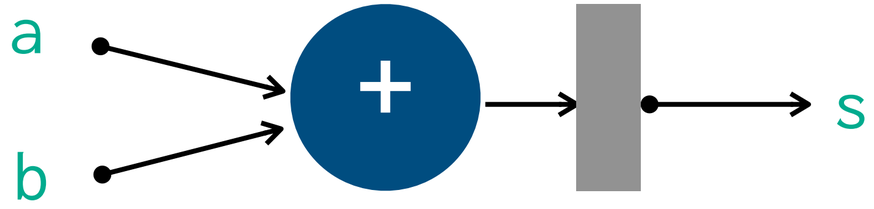
Combinational

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




  assert_eq(DUT.s, s);
  step();
}
```

Pipelined (naïve)

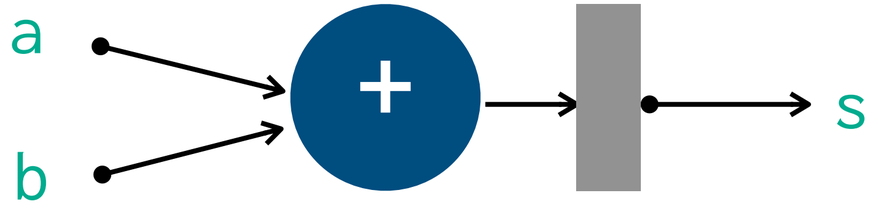
```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
+ step();
+ fork();
  assert_eq(DUT.s, s);
  step();
}
```

A naïve attempt at a pipelined adder




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

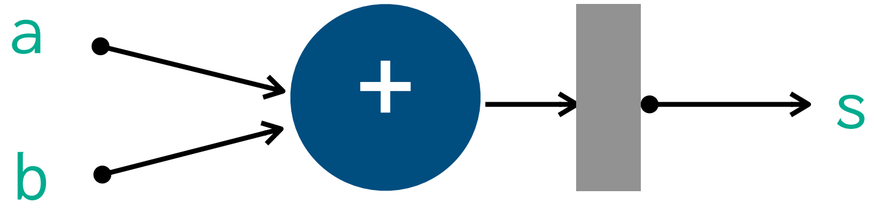
A naïve attempt at a pipelined adder




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

`fork()`: allows another transaction to begin concurrently in the same cycle (pipelining!)

A naïve attempt at a pipelined adder




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

2nd `step()` for a total latency of 2 cycles

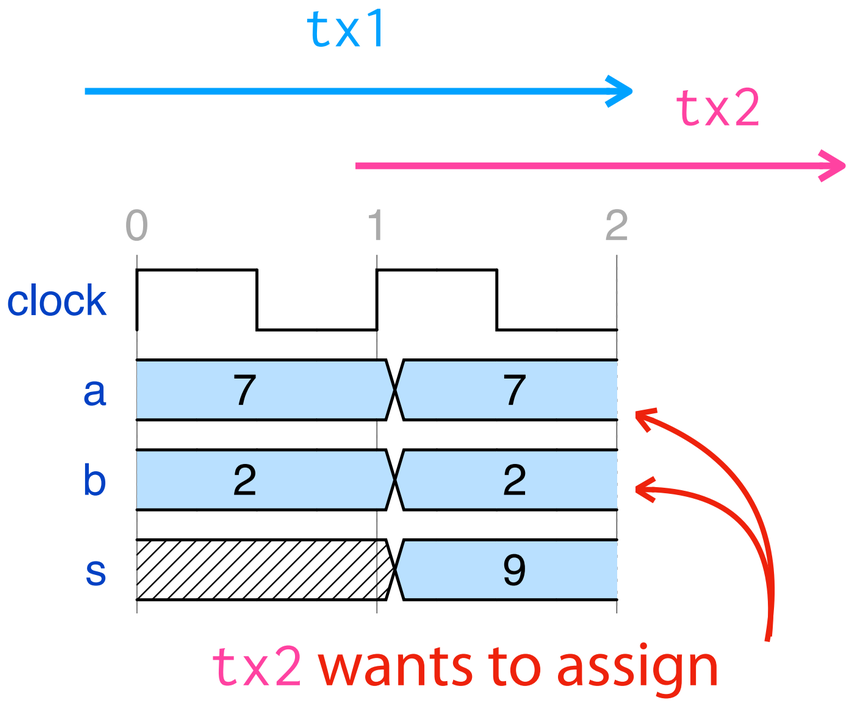
A naïve attempt at a pipelined adder

Pipelined (naïve)

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

Problem: tx2 wants to drive values onto DUT.a & DUT.b, but tx1 still has control over these ports!

```
tx1: add(7, 2, 9)
tx2: add(2, 4, 6)
```

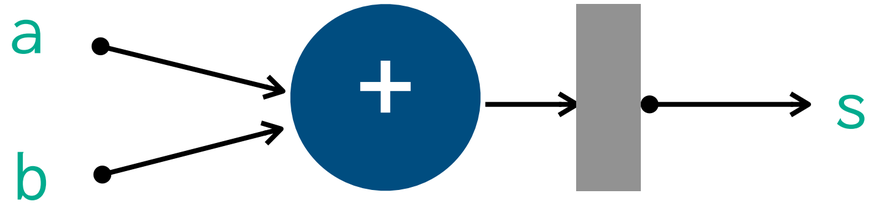


tx2 wants to assign

```
DUT.a := 2; DUT.b := 4
```

here but is unable to do so!

A pipelined adder, take two



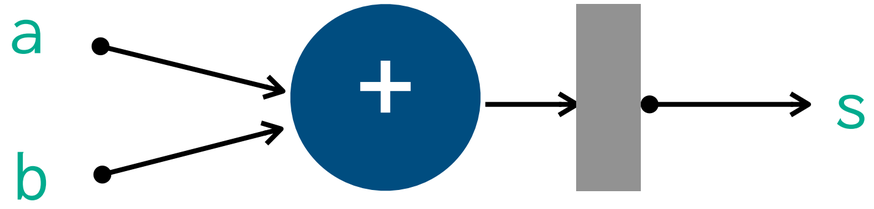
Combinational

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



  assert_eq(DUT.s, s);
  step();
}
```

Pipelined

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

A pipelined adder, take two



Combinational

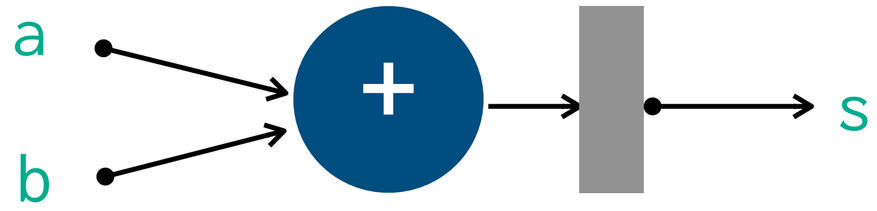
```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;




  assert_eq(DUT.s, s);
  step();
}
```

Pipelined

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


new!

A pipelined adder

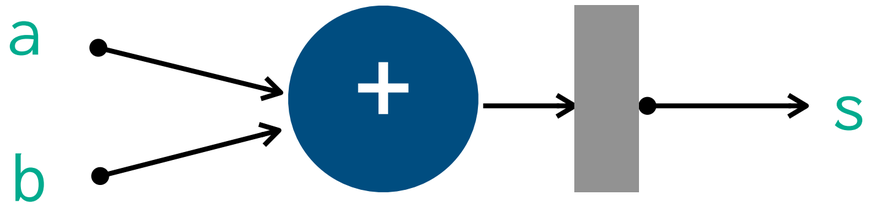



```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  step();
  DUT.a := X; DUT.b := X;
  fork();
  assert_eq(DUT.s, s);
  step();
}
```

`DontCare` assignments: release constraints on ports

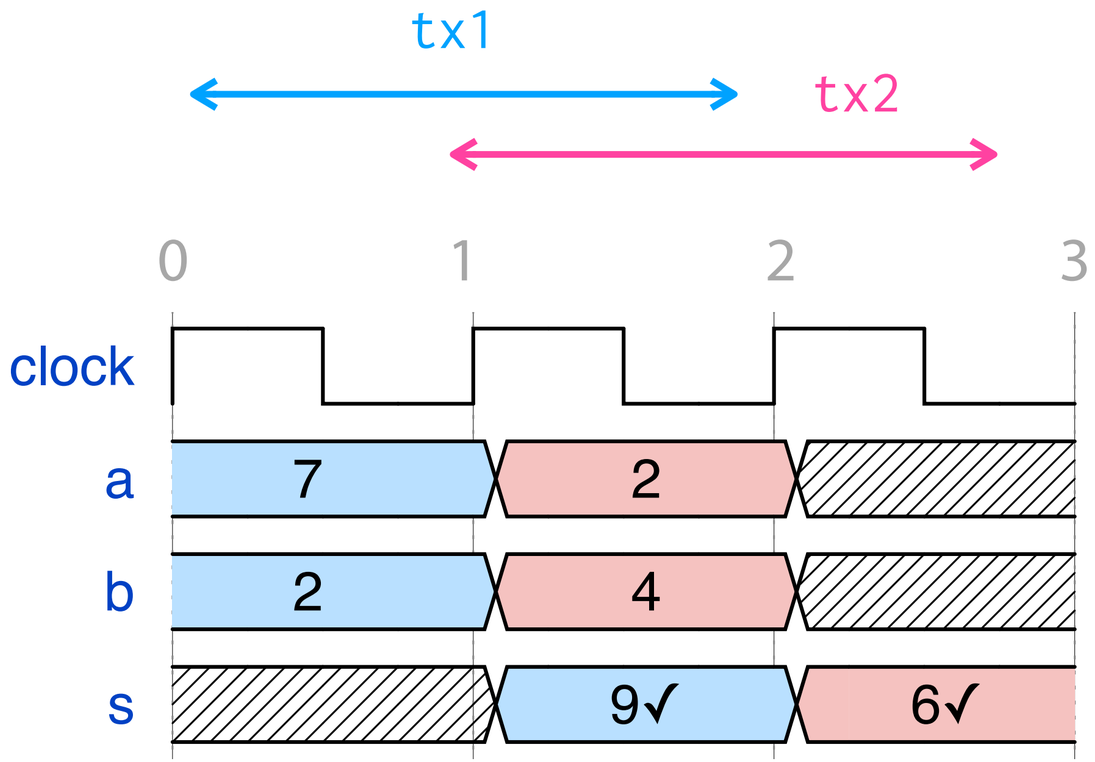
(Value on ports `DUT.a` & `DUT.b` no longer matter)

A pipelined adder




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

```
tx1: add(7, 2, 9)
tx2: add(2, 4, 6)
```



Second example:
A Ready-Valid Handshake in Paso

A Ready-Valid Handshake in Paso

Parameterized over `DUT`s (Designs Under Test) that implement the `ReadyValid` interface

```
prot send_data<DUT: ReadyValid>(data: u8) {



}
```

A Ready-Valid Handshake in Paso

```
prot send_data<DUT: ReadyValid>(data: u8) {
    DUT.valid := 1;
}
```

Indicate that we have
semantically meaningful data to send

A Ready-Valid Handshake in Paso

```
prot send_data<DUT: ReadyValid>(data: u8) {
    DUT.valid := 1;
    DUT.data := data;

}
```

Indicate that the `data` parameter is the payload to be sent

A Ready-Valid Handshake in Paso

```
prot send_data<DUT: ReadyValid>(data: u8) {
    DUT.valid := 1;
    DUT.data := data;
    while (DUT.ready ≠ 1) {
      step();
    }

}
```

Wait until `ready` becomes 1
(i.e. till the receiver becomes ready)

A Ready-Valid Handshake in Paso

```
prot send_data<DUT: ReadyValid>(data: u8) {
    DUT.valid := 1;
    DUT.data := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

The actual data transfer takes one more clock cycle

A Ready-Valid Handshake in Paso

Simple imperative semantics, just like software!

```
prot send_data<DUT: ReadyValid>(data: u8) {
    DUT.valid := 1;
    DUT.data := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

Interpreter Design

Goal: address common pain-points with ad-hoc Verilog testbenches

- Race conditions
- Sampling transient values (“glitches”)

Detecting conflicting assignments in concurrent transactions

At the end of each cycle, a transaction can only observe a unique value for each DUT port

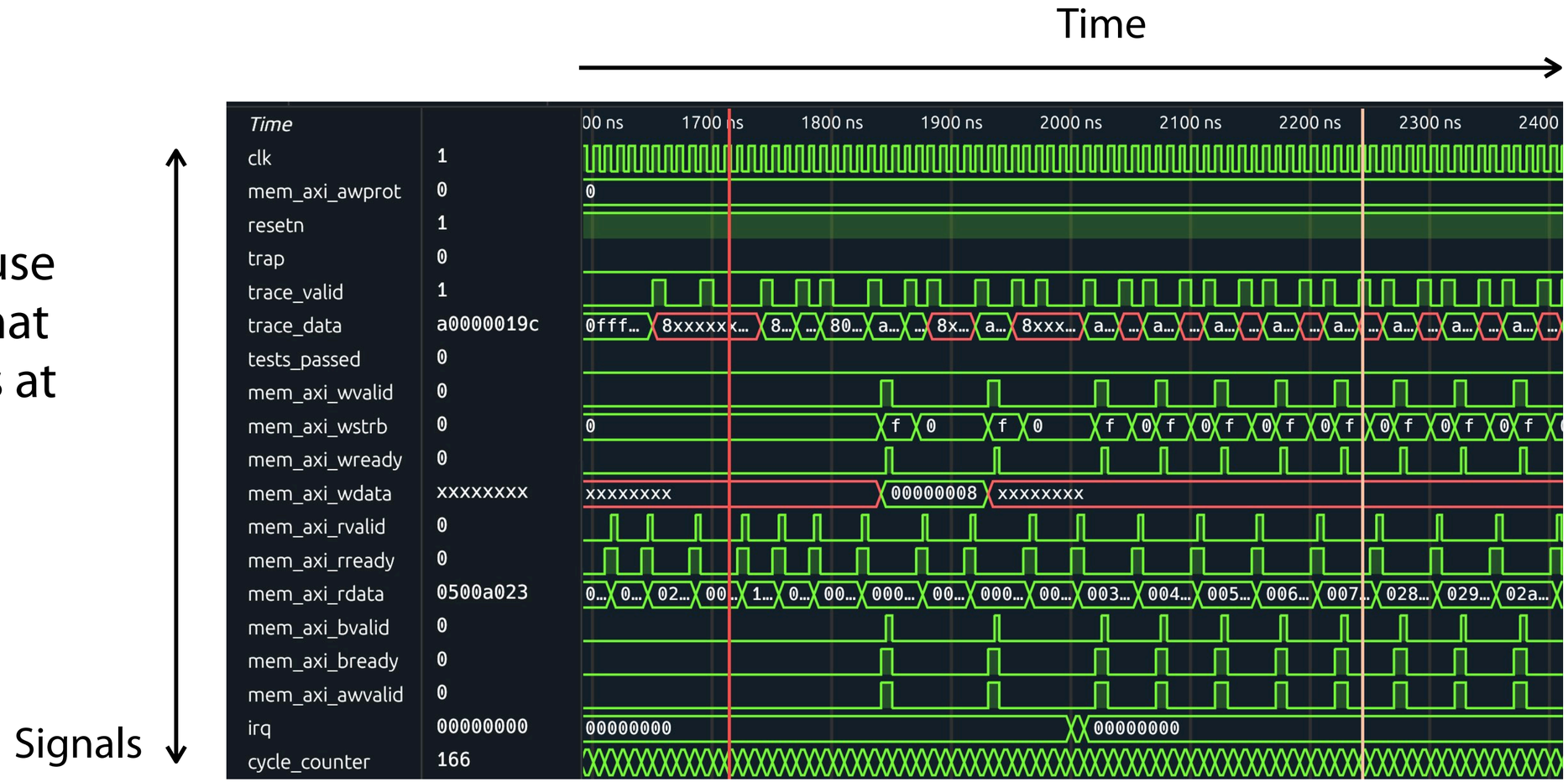
	Transaction 1	Transaction 2
✓	`DUT.a := 5`	`DUT.a := X`
✓	`DUT.a := X`	`DUT.a := X`
X	`DUT.a := 5`	`DUT.a := 6`

Implementation: tracks combinational dependencies

Reconstructor

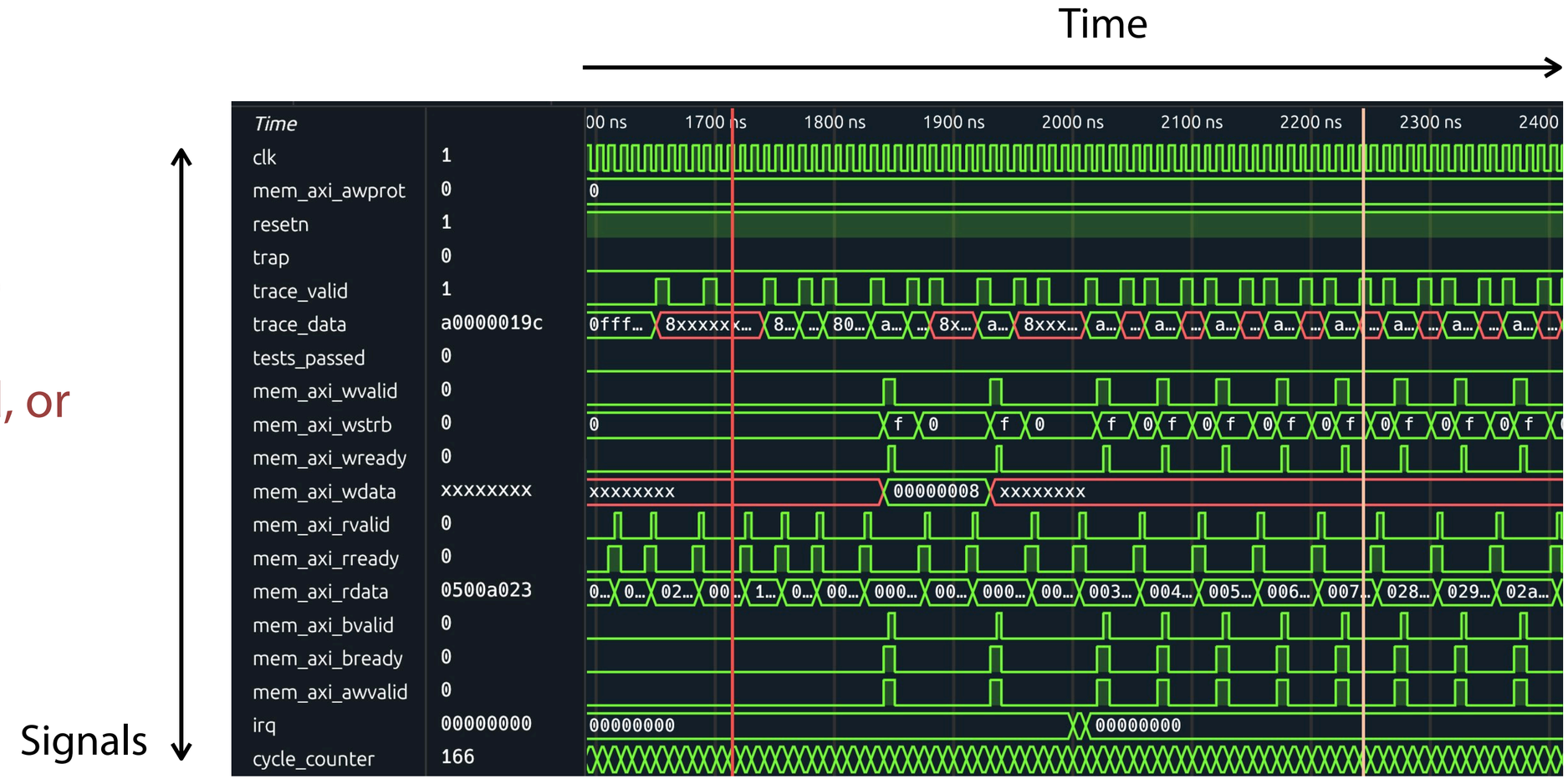
Motivation: Debugging hardware is hard

Hardware designers use ***waveform viewers*** that display signals' values at each cycle

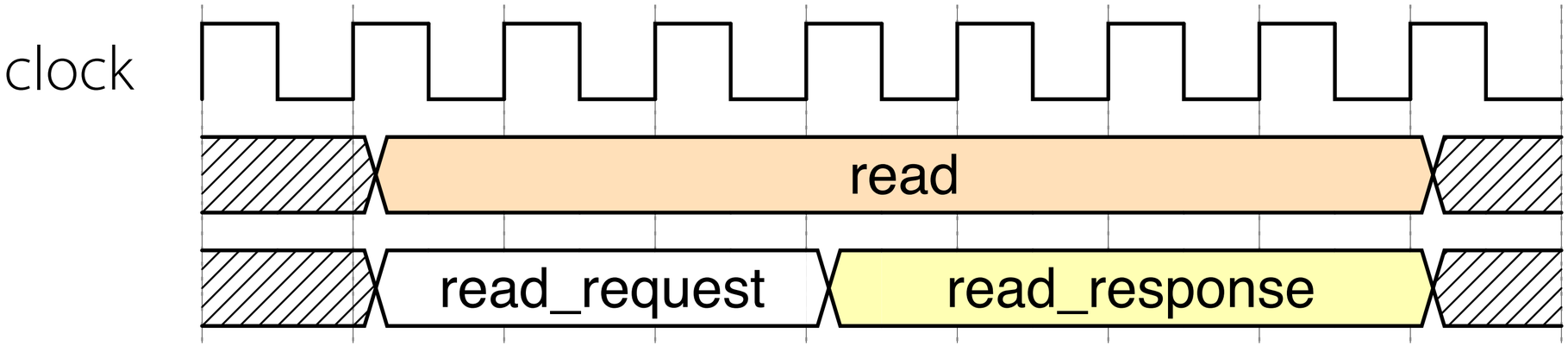


Motivation: Debugging hardware is hard

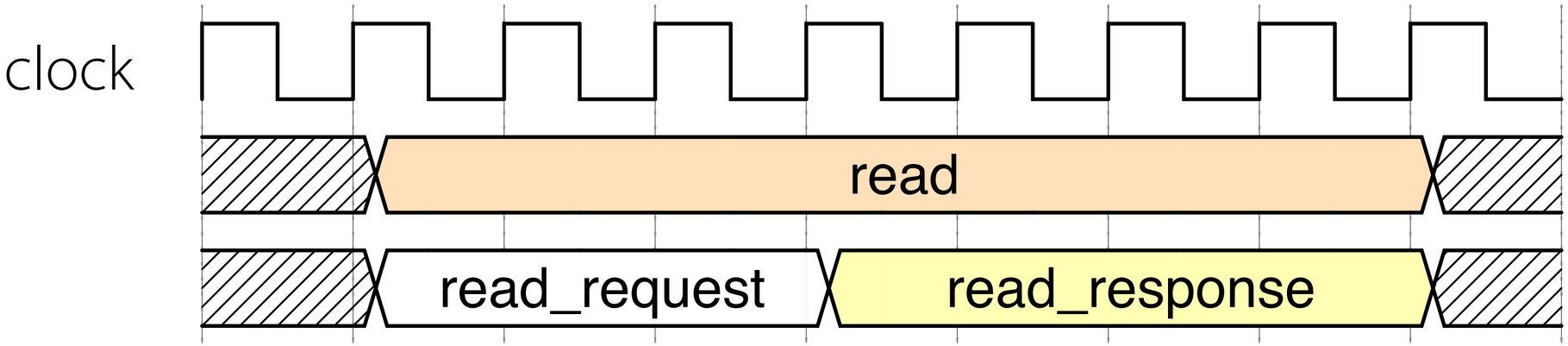
Problem: hard to identify ***when*** a transaction occurred, or ***how long*** it took!



What if we had waveform viewers that looked like this instead?



Problem: need to *reconstruct* transactions from signals



Our reconstructor does this!

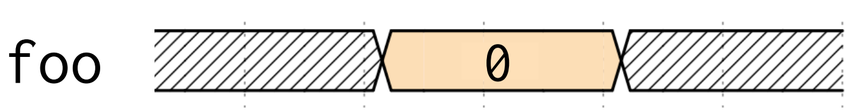
How does the reconstructor work?

Assignments as Constraints

`foo := 0`

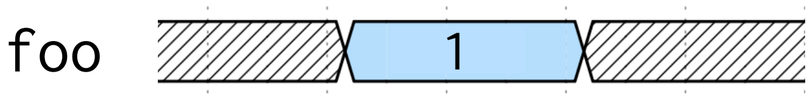
means

"Check if the current waveform value of `foo` is equal to `0`"



Matches!

✓



Inconsistent!

X

Treat `foo == 0` as a constraint for the rest of the protocol
(or until `foo` is reassigned)

Assignment in protocol body

Constraint in reconstructor

`LHS := RHS` ⟶ `RHS` = waveform value of **LHS** at the current clock cycle

`foo := 0` ⟶ `0 = waveform(foo)`

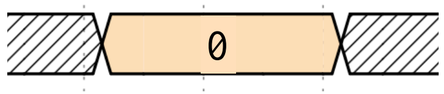
`foo := a` ⟶ `a = waveform(foo)`

Assignment in protocol body

Constraint in reconstructor

`LHS := RHS` ⟶ `RHS` = waveform value of **LHS** at the current clock cycle

`foo := 0` ⟶ `0 = 0`

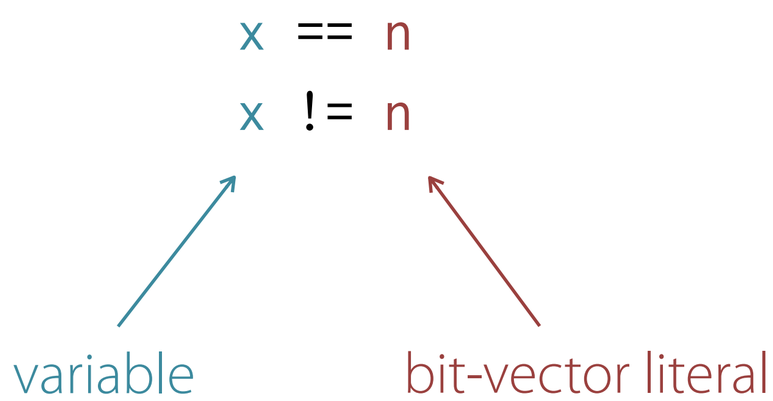
`foo` 

`foo := a` ⟶ `a = 0`

No SMT solvers needed!

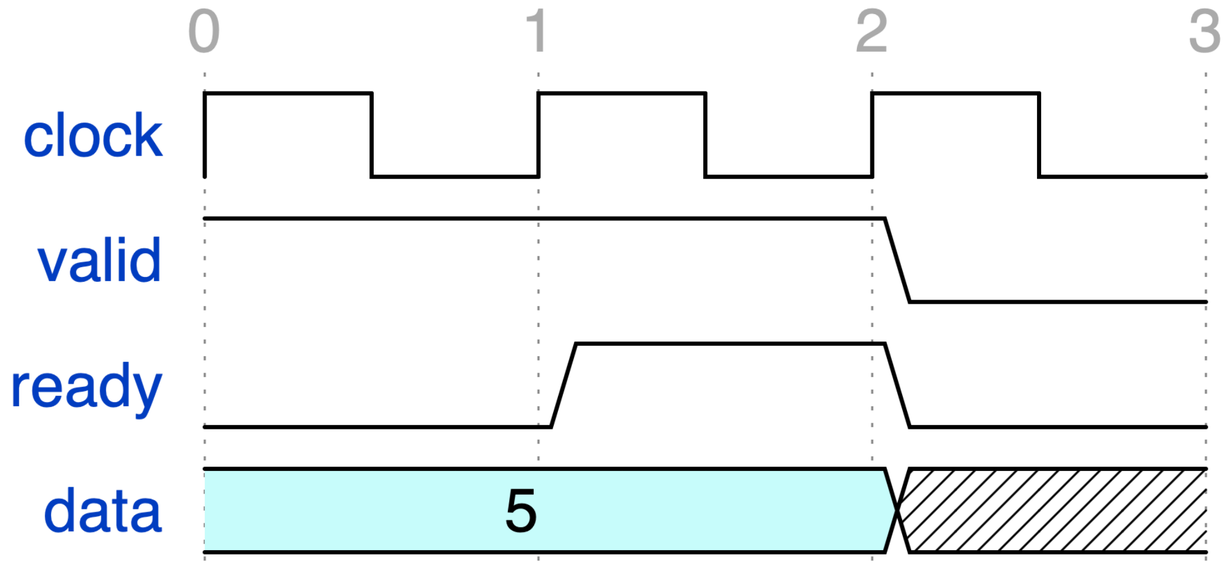
All constraints in our DSL are equality constraints where one side is a constant

$\Longrightarrow$ can be solved efficiently (by inspecting waveform) without relying on solvers



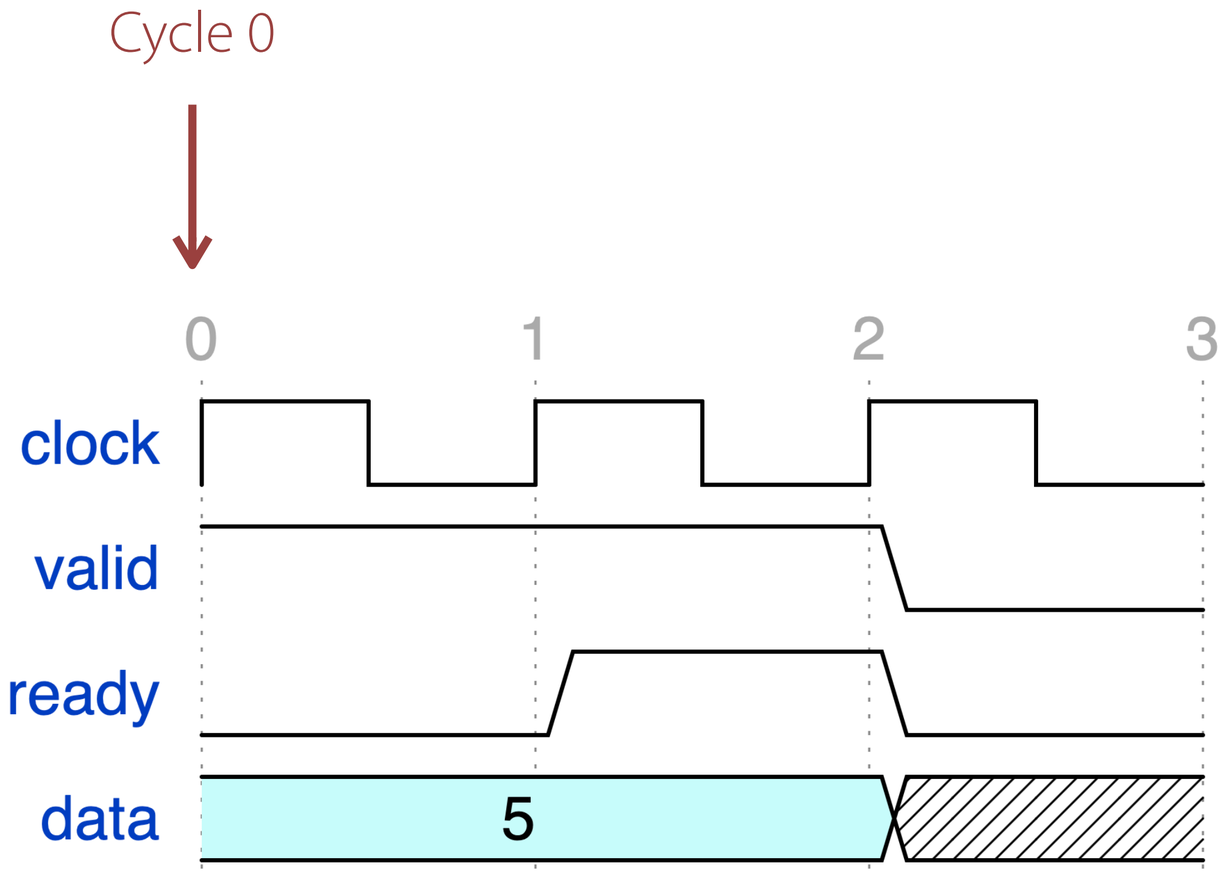
Idea: symbolically execute protocols & compare against waveform

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```



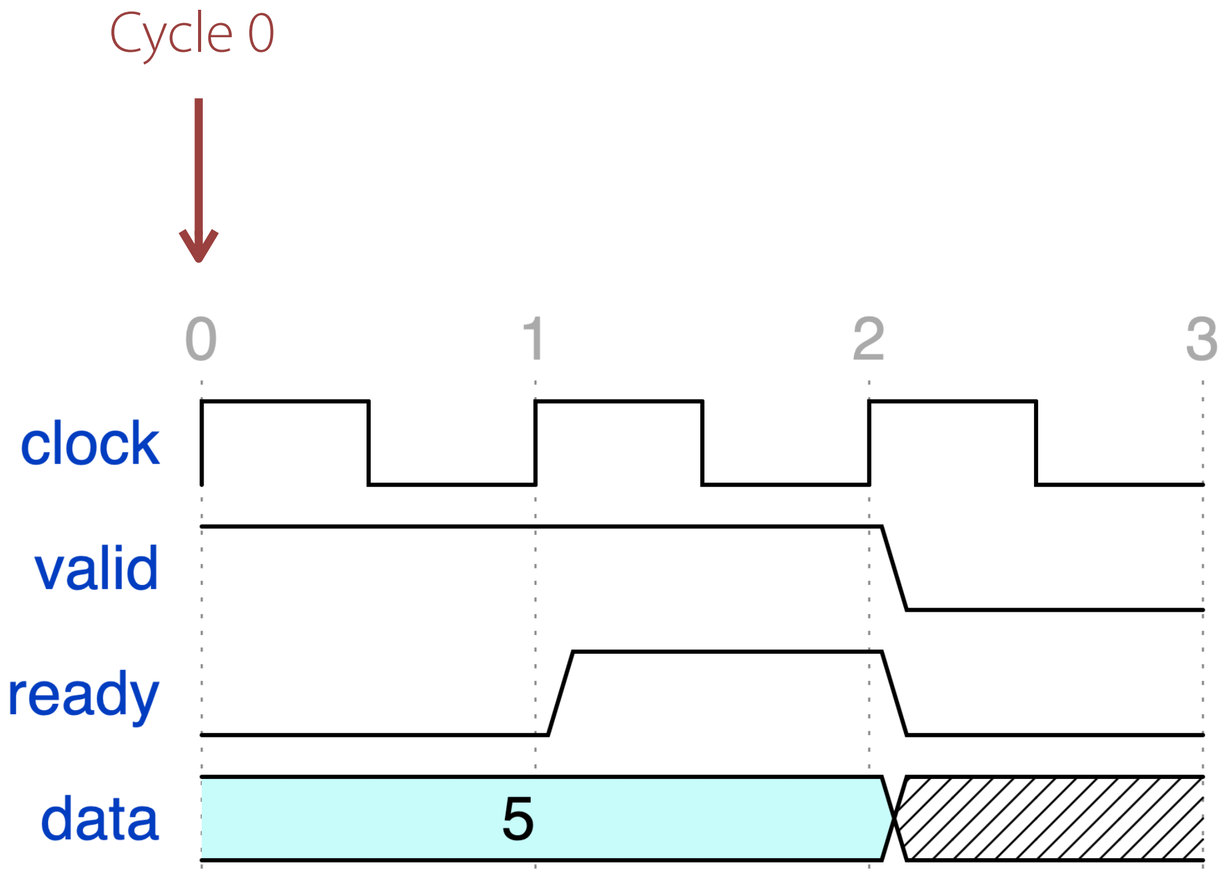
Constraints: ∅

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```



Constraints: ∅

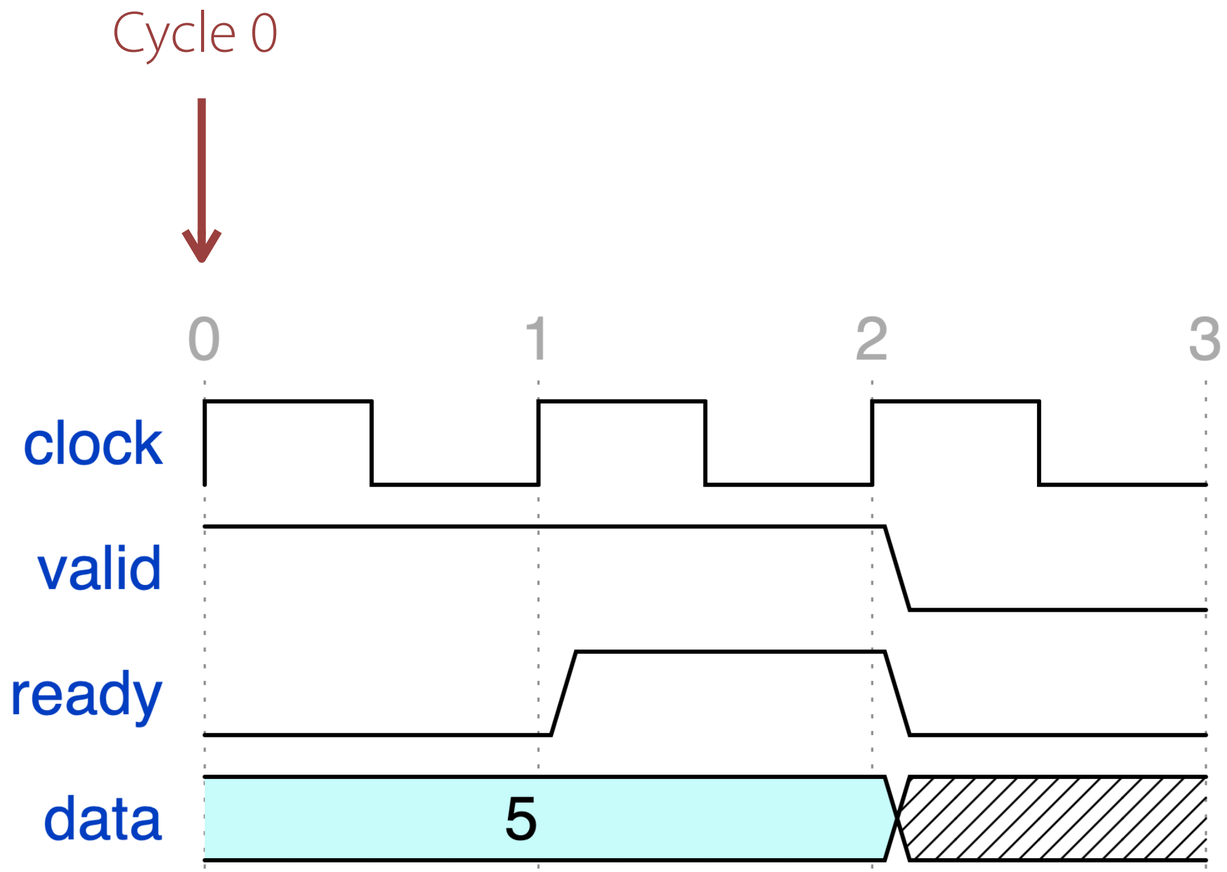
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```



Constraints: ∅

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

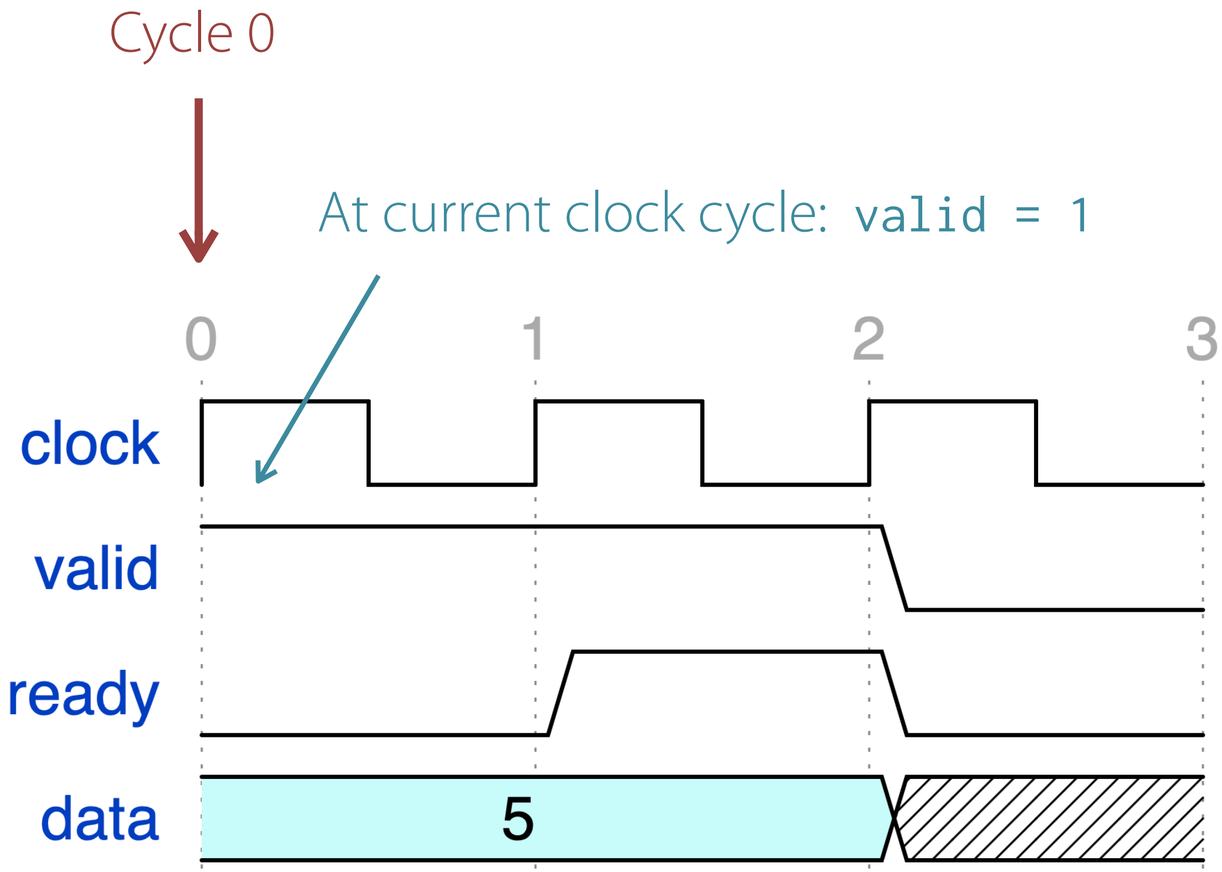
Check if current waveform value of `valid =? 1`



Constraints: ∅

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

Check if current waveform value of `valid =? 1`



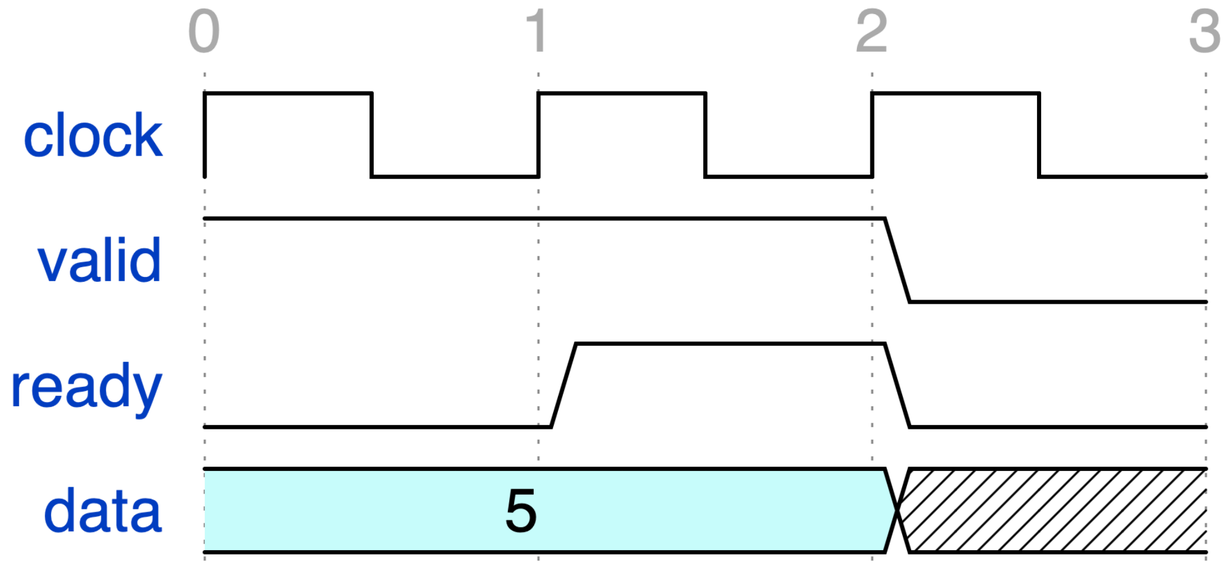
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Check if current waveform value of `valid =? 1`

Constraints: `{ 1 = 1 }`

Cycle 0

`DUT.valid := 1` is consistent w/ waveform!
Add a new constraint.



```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

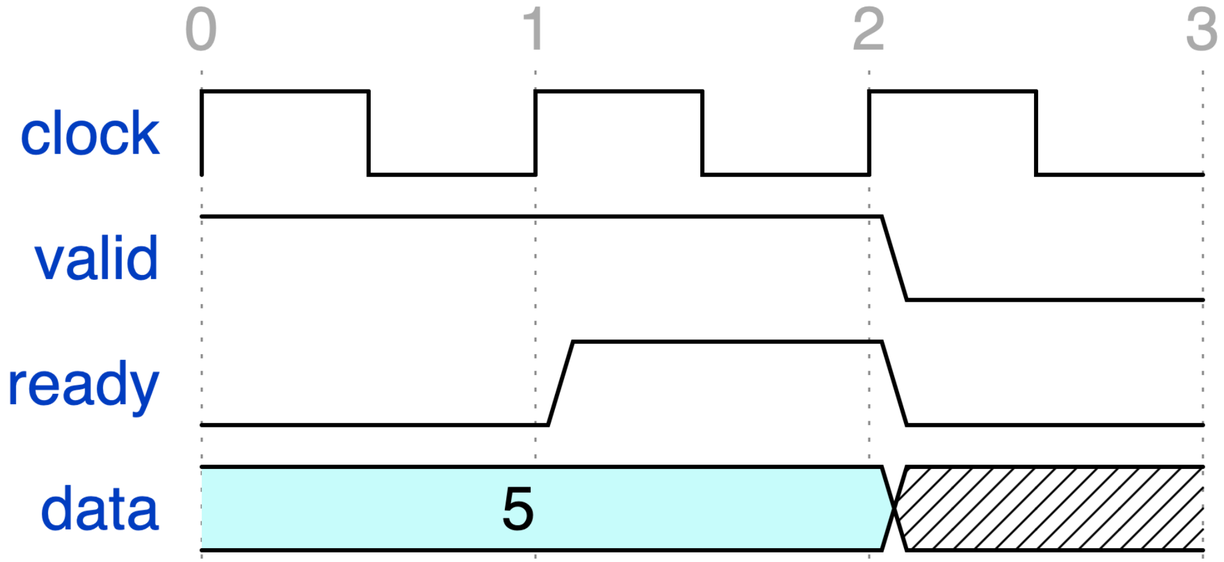
Check if current waveform value of `valid` =? 1

Constraints: { 1 = 1 }

Cycle 0

RHS of the stmt
`DUT.valid := 1`

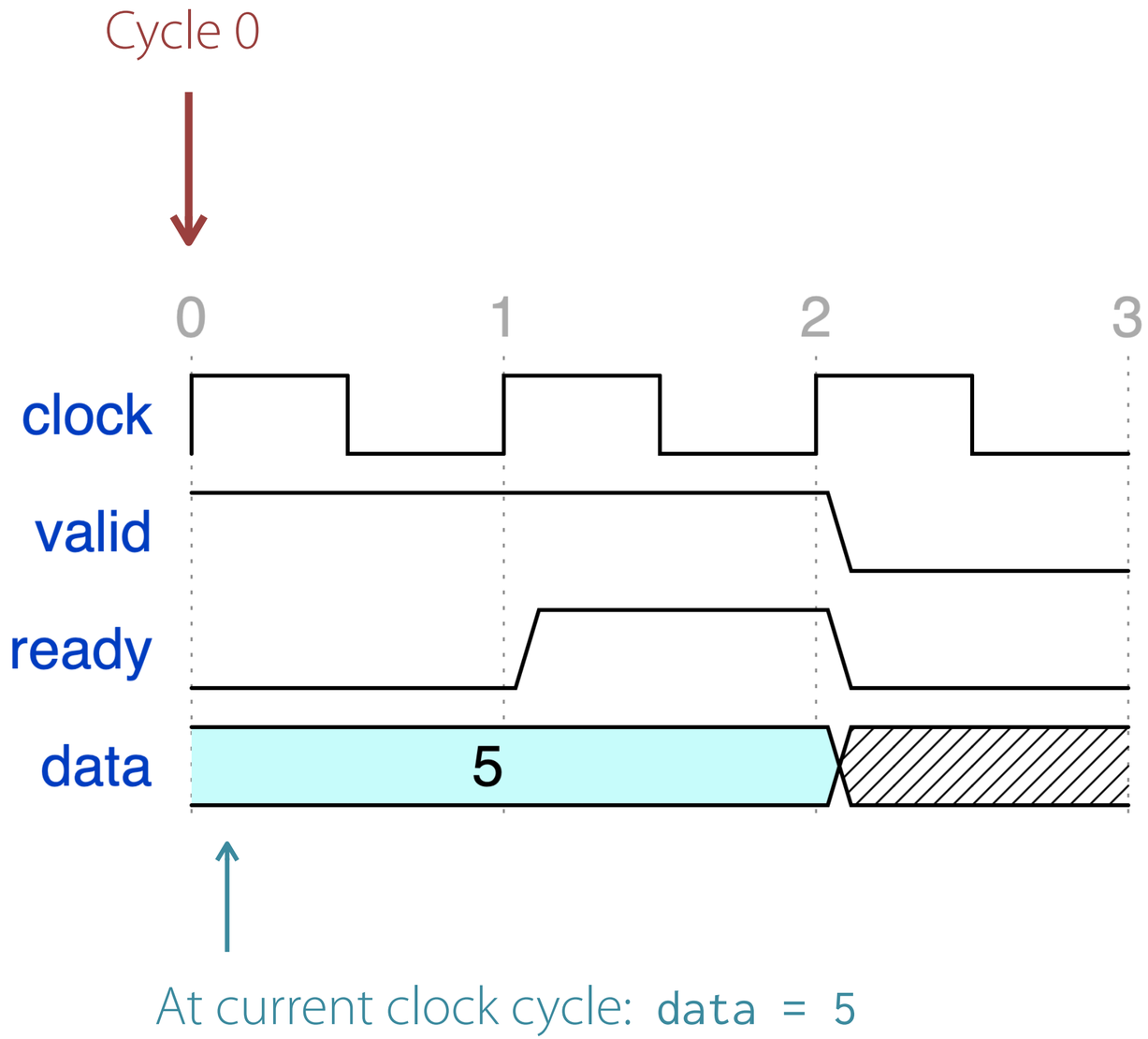
Waveform value of `valid` @ current clock cycle



Constraints: `{ 1 = 1 }`

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Look up current value of RHS (`data`) from waveform

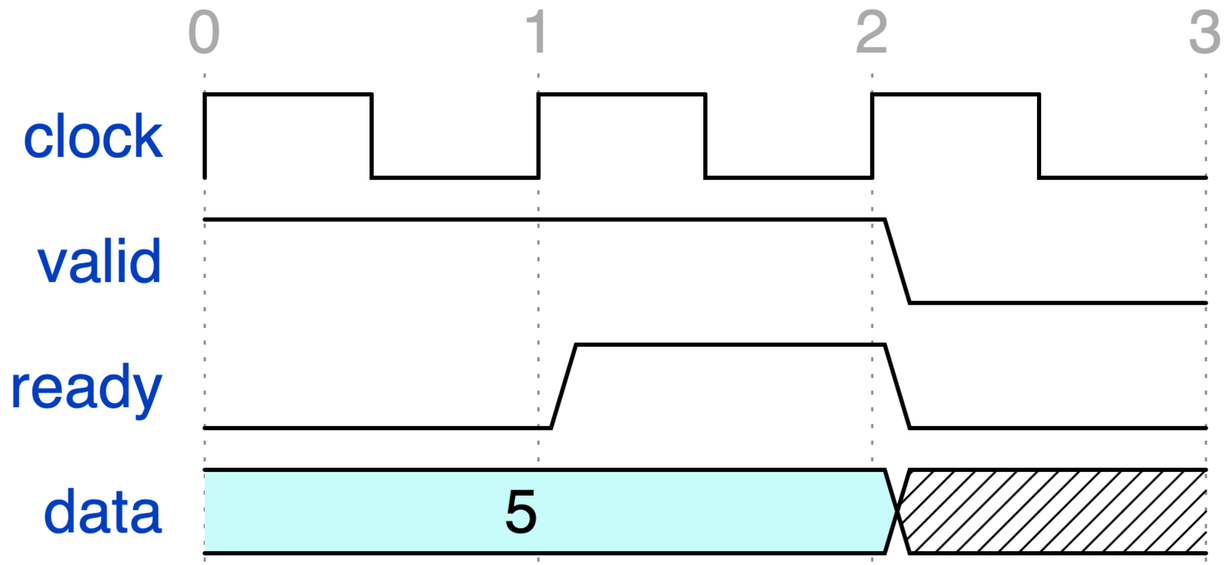


```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: { 1 = 1, data = 5 }

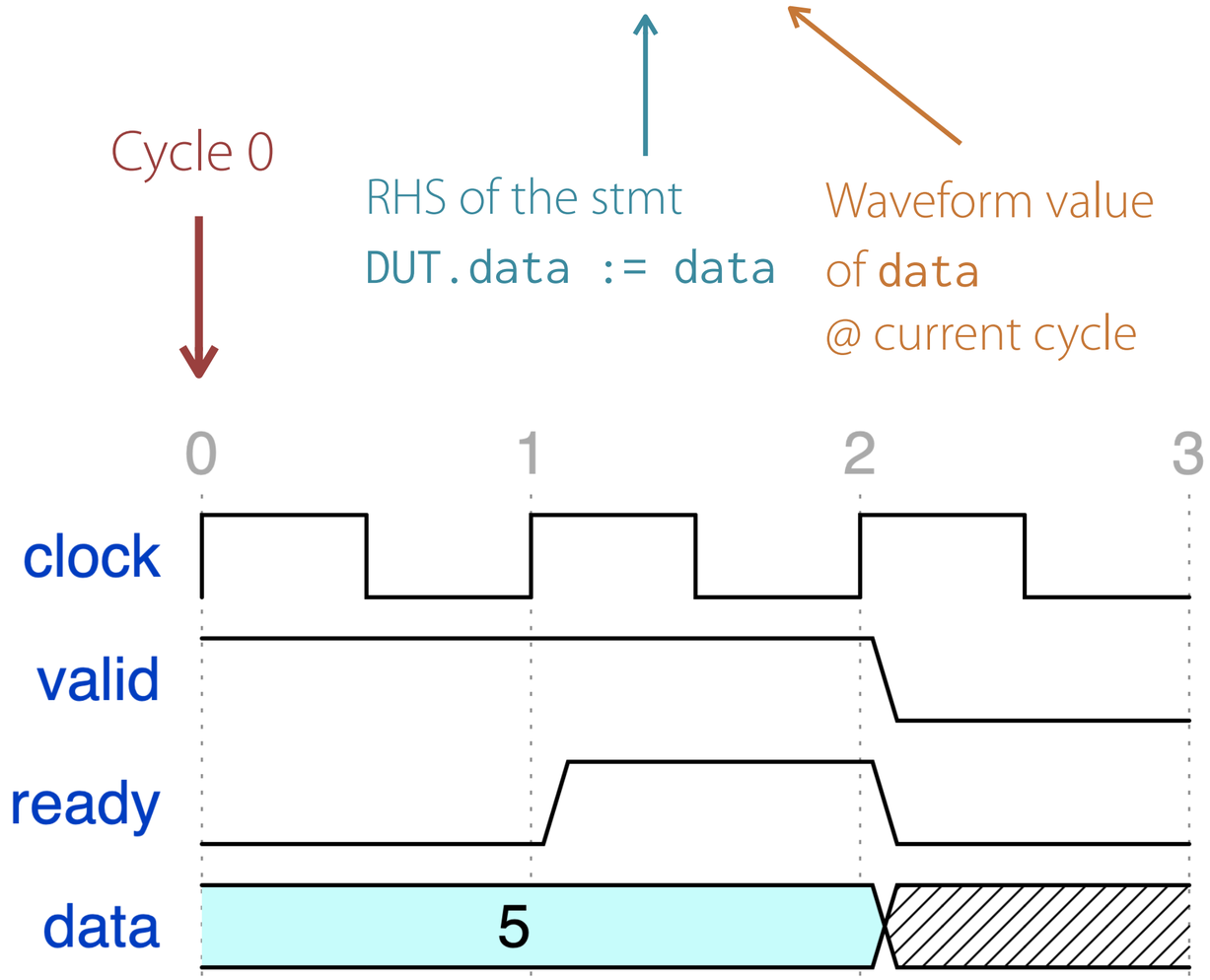
Add a new constraint `data = 5`

Cycle 0



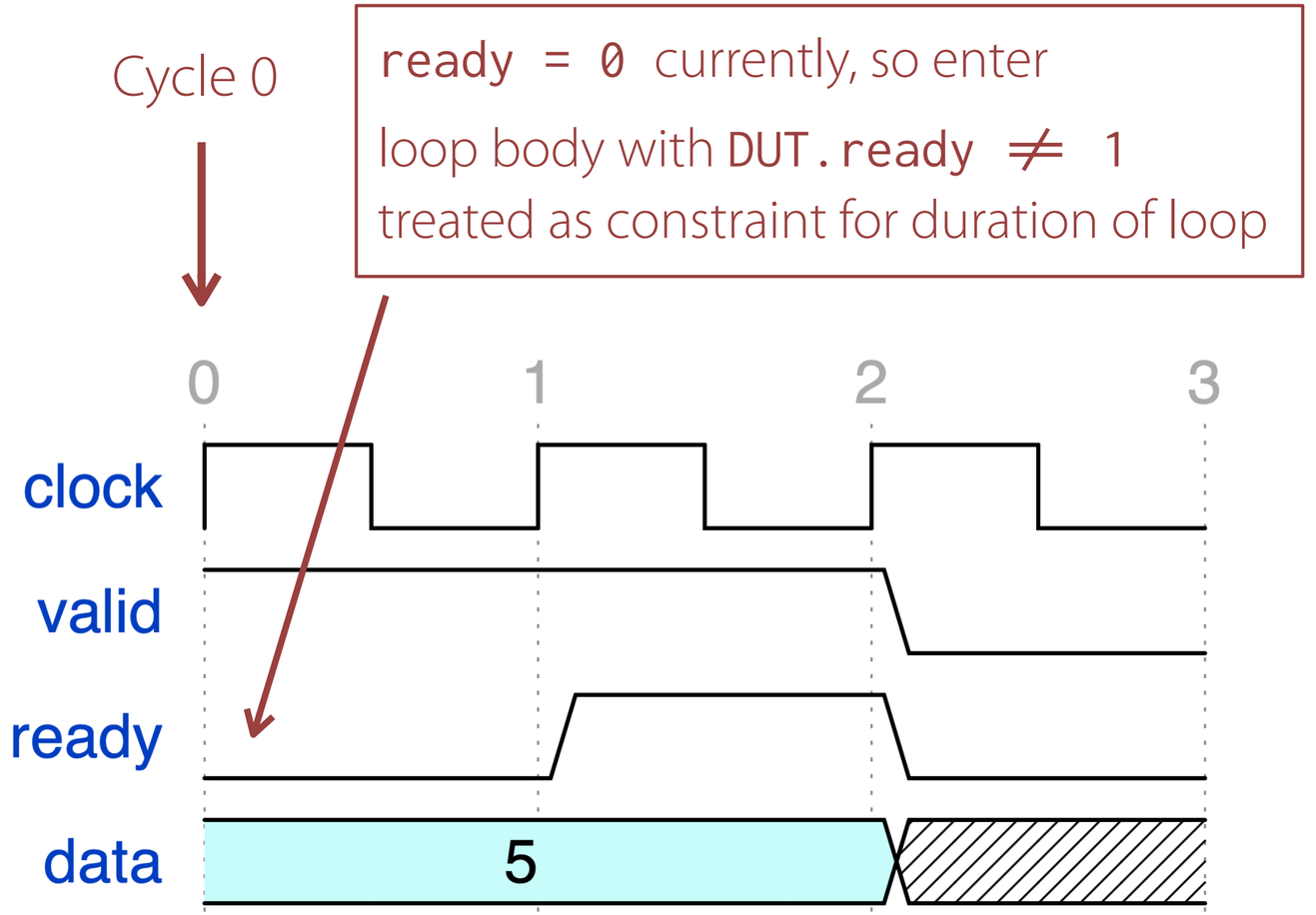
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: { 1 = 1, data = 5 }



```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: `{ 1 = 1, data = 5 }`

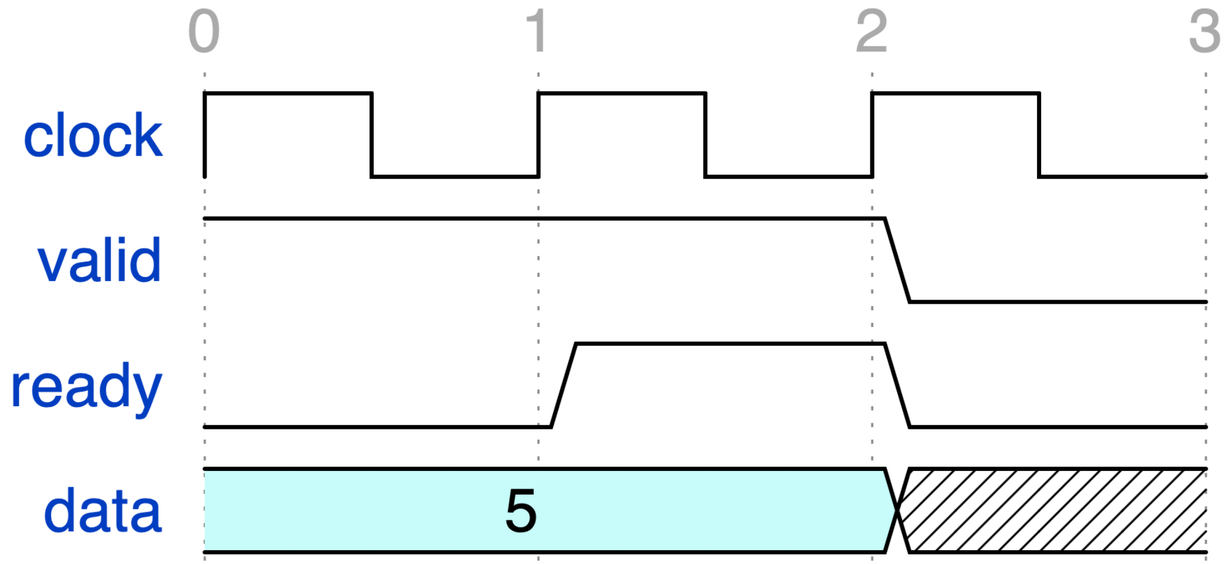


```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

Cycle 0

Add a new constraint 1 ≠ 0 representing the loop guard



```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

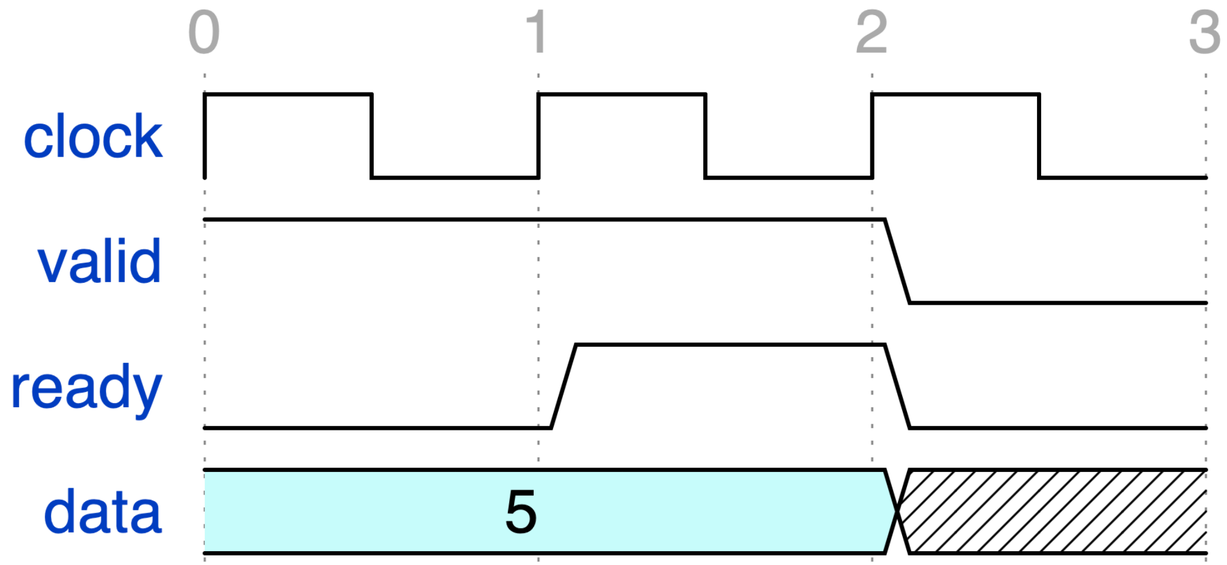
Cycle 0

RHS of conditional

DUT.ready ≠ 1

Waveform value of

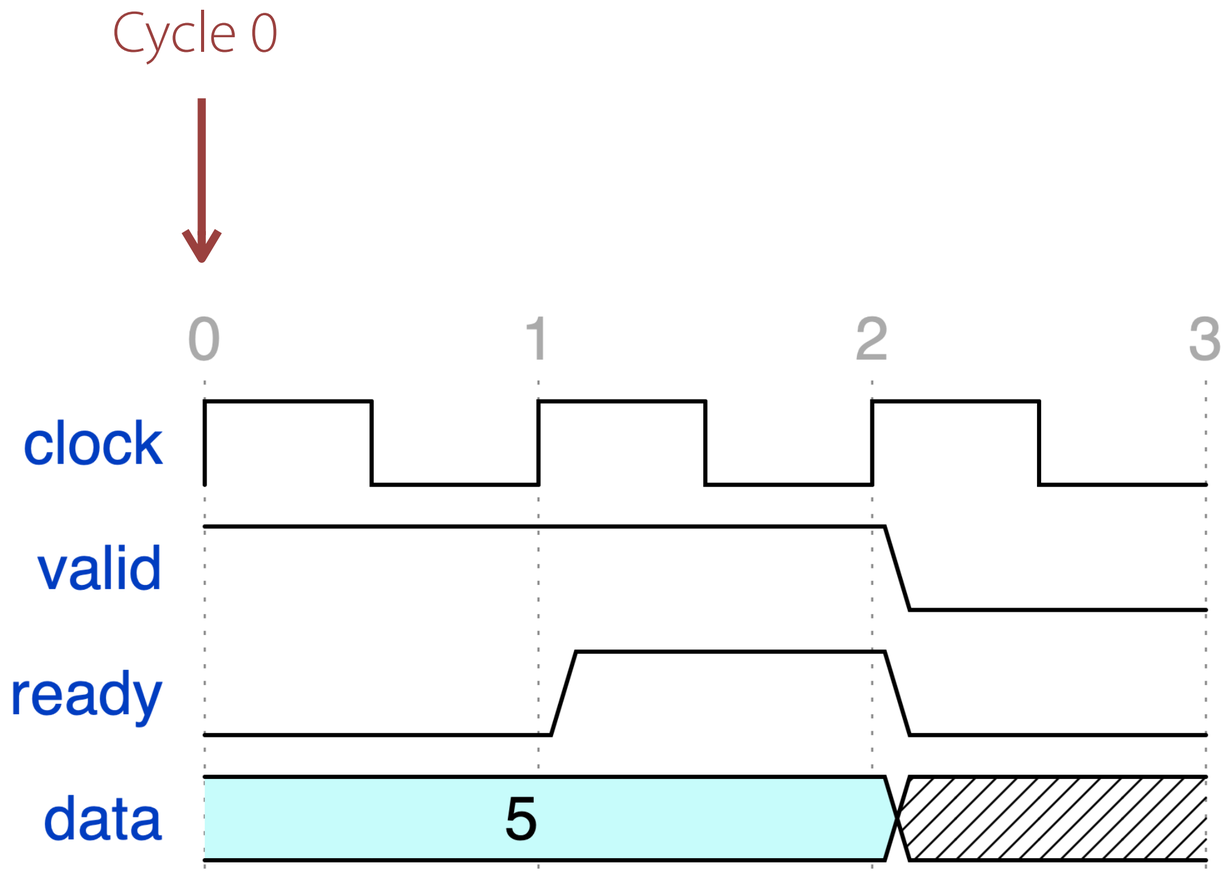
ready @ current cycle



Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

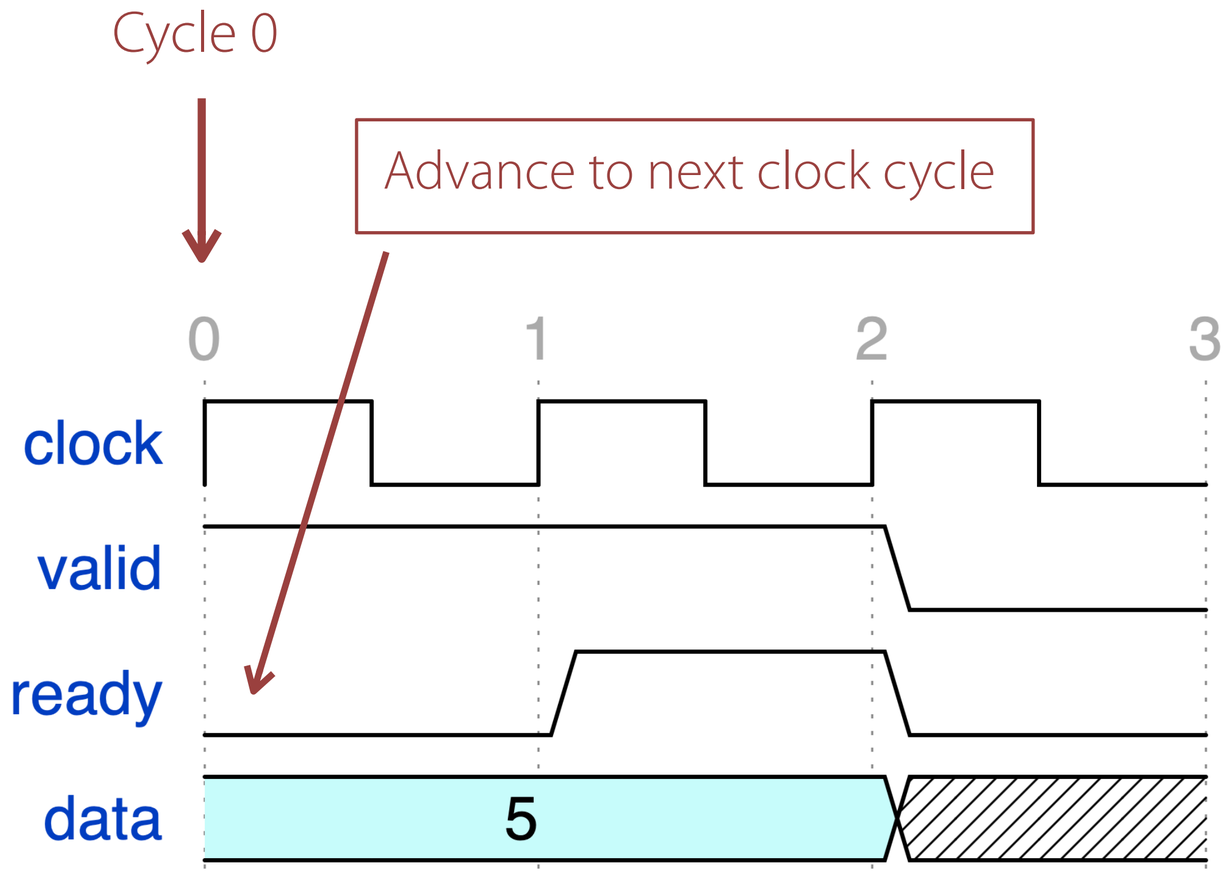
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Evaluate loop body



Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

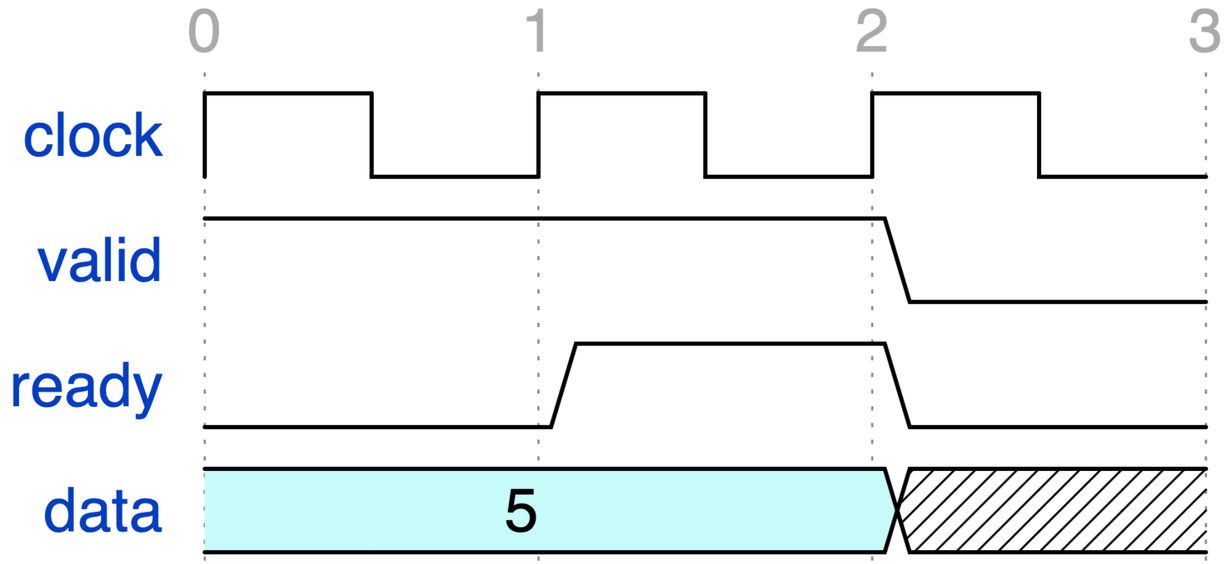


Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Cycle 1

Evaluate loop guard again:
We now have `ready = 1`,
so exit loop & remove
constraint for guard

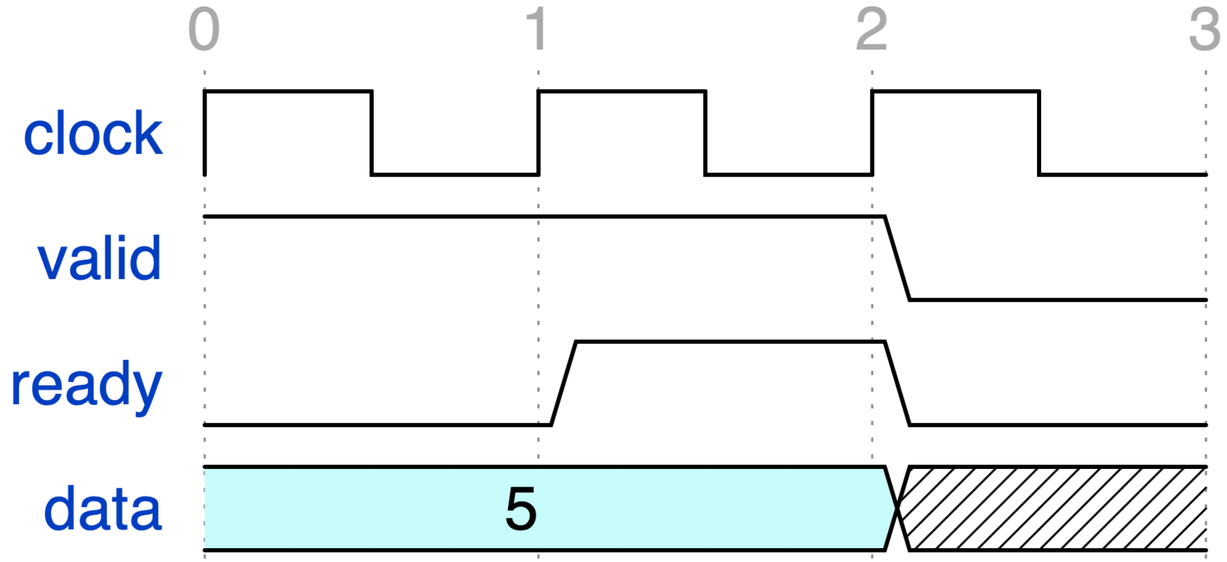


```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

Constraints: `{ 1 = 1, data = 5 }`

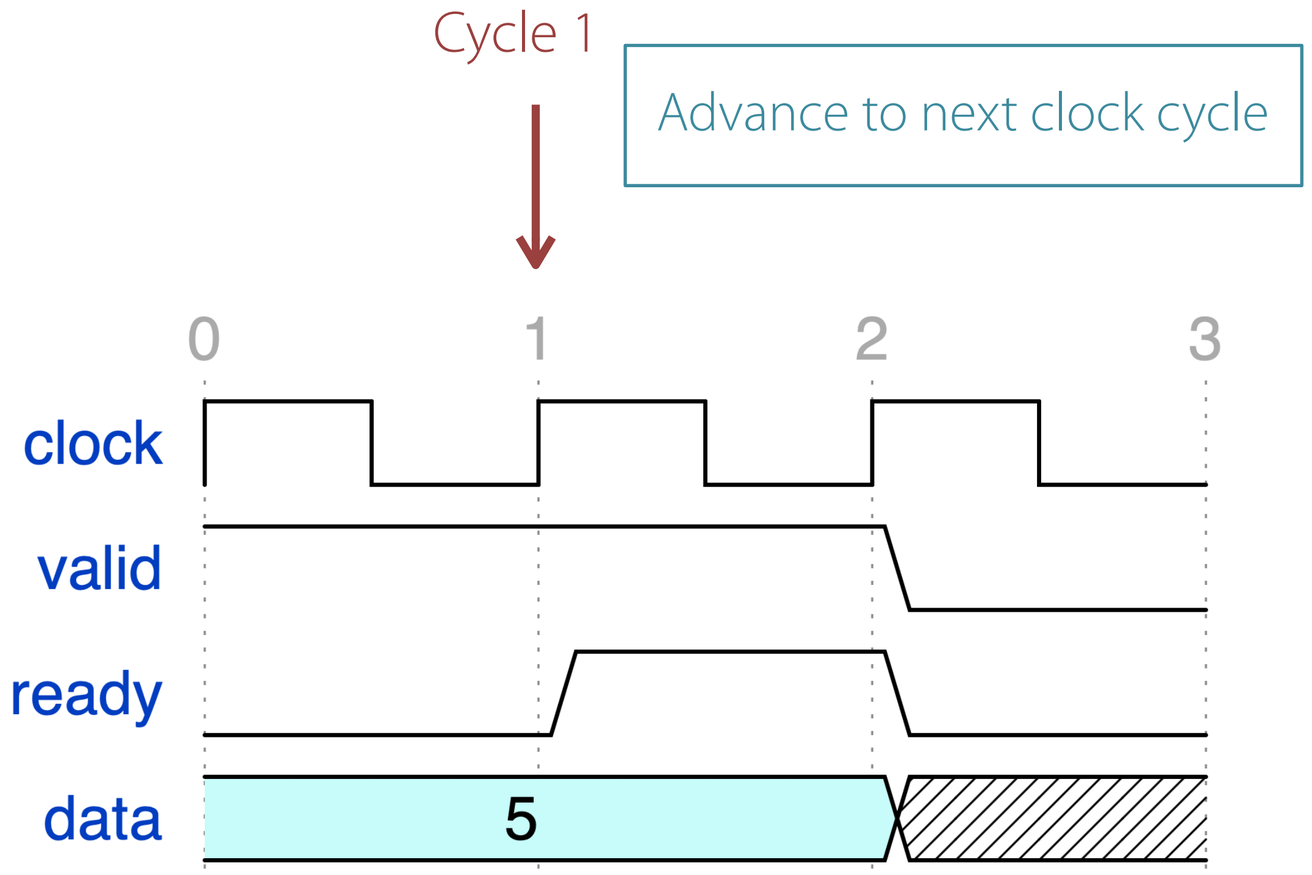
Cycle 1

Constraint `0 ≠ 1` removed



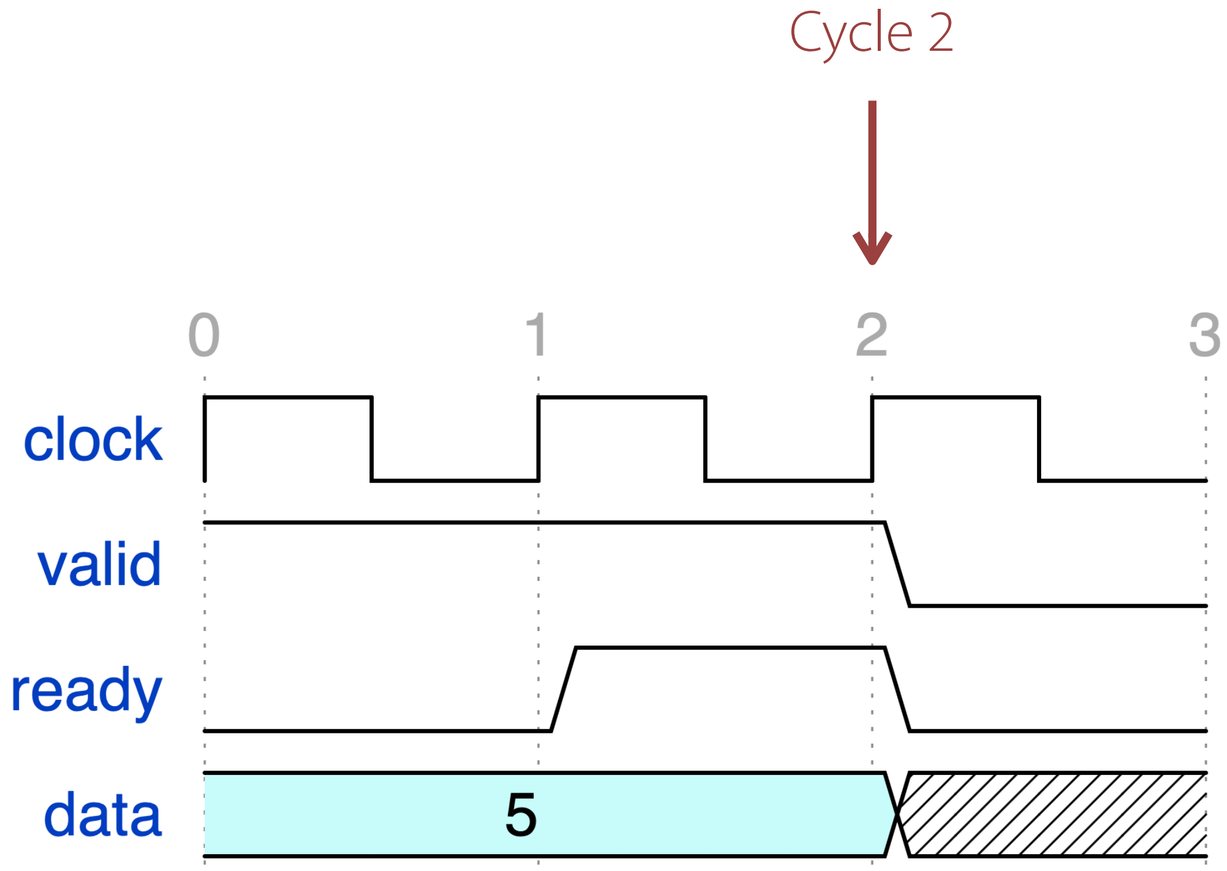
Constraints: `{ 1 = 1, data = 5 }`

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```



Constraints: `{ 1 = 1, data = 5 }`

```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

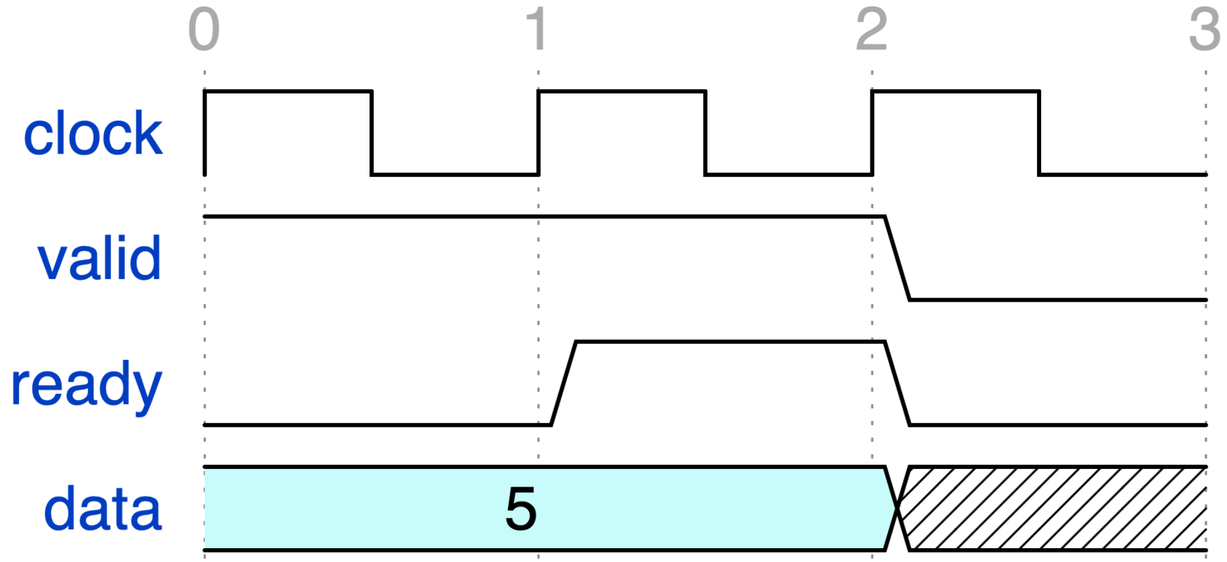


```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: `{ 1 = 1, data = 5 }`

Finished symbolically executing `send_data`!

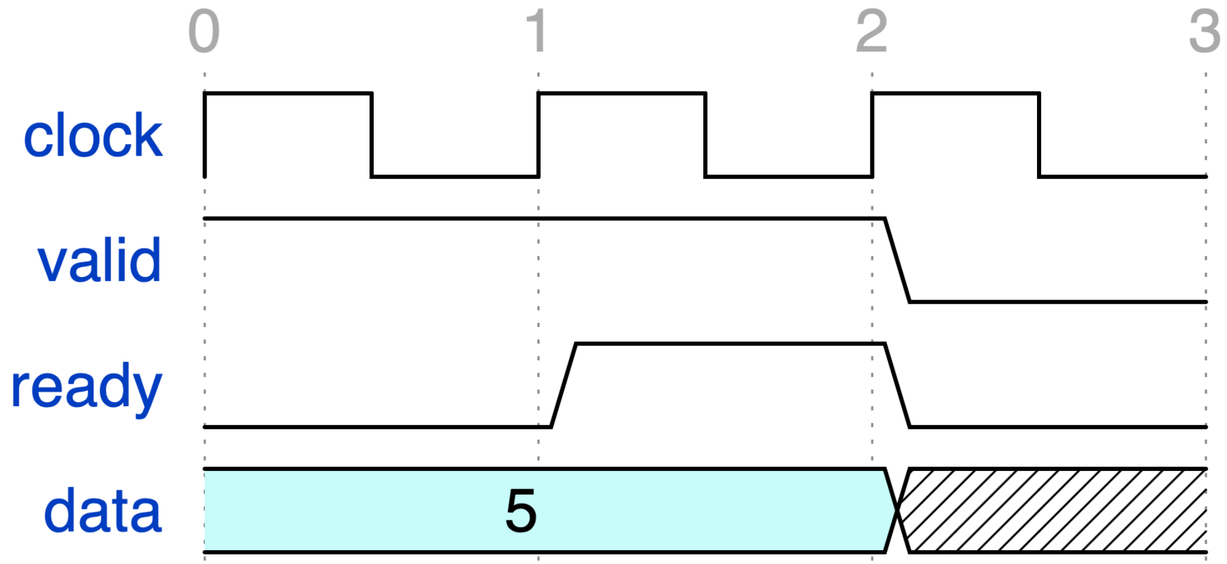
Cycle 2



```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

Inferred transaction trace:

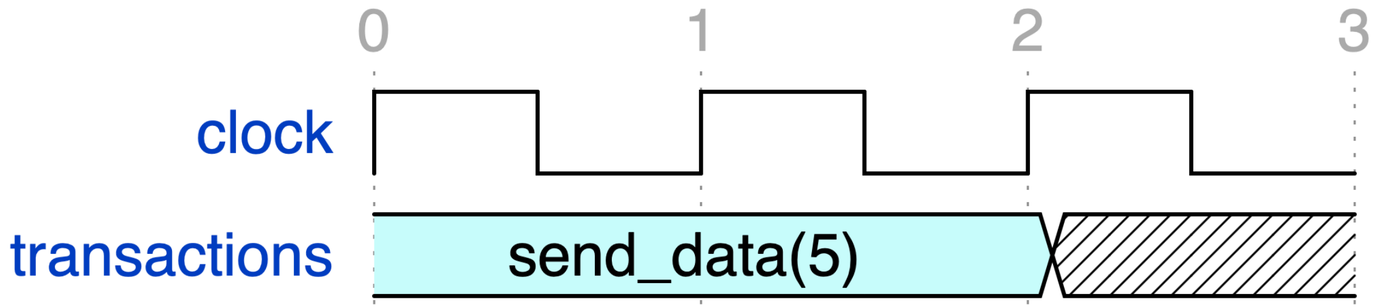
```
send_data(5); // cycles 0-2
```



```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

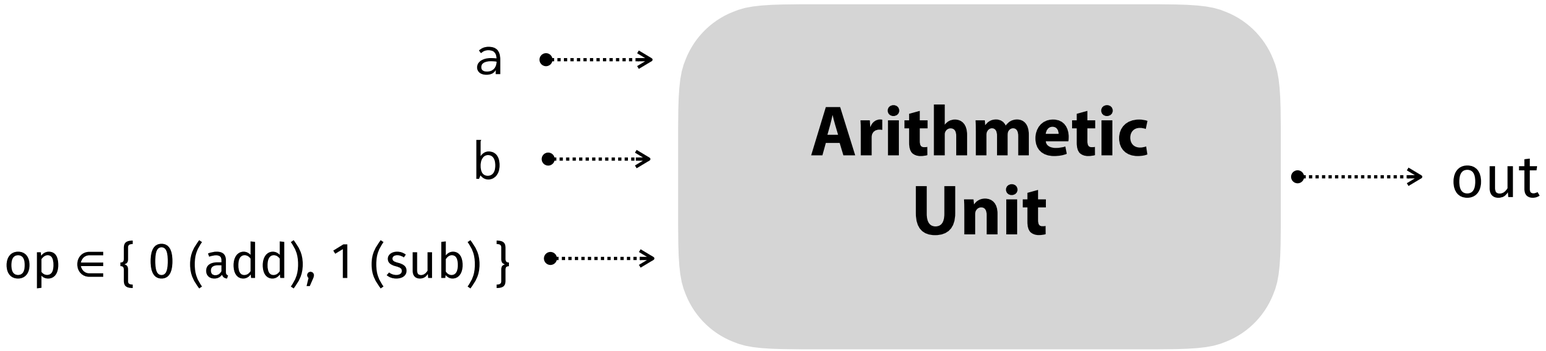
Inferred transaction trace:

```
send_data(5); // cycles 0-2
```



A reconstructor example with multiple protocols

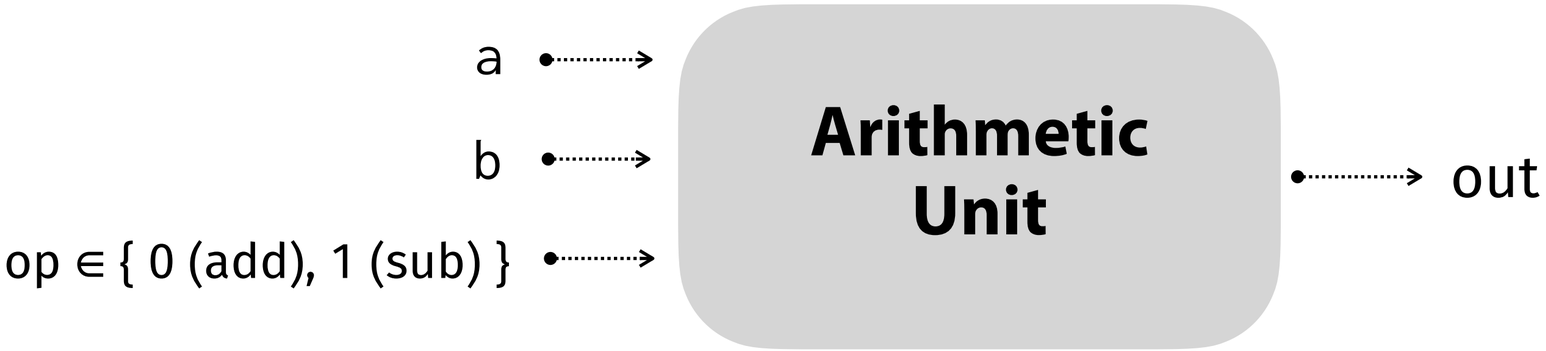
Example: Arithmetic Unit




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

```
prot sub<DUT: ALU>(a, b, out) {
  DUT.a := a; DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

Example: Arithmetic Unit



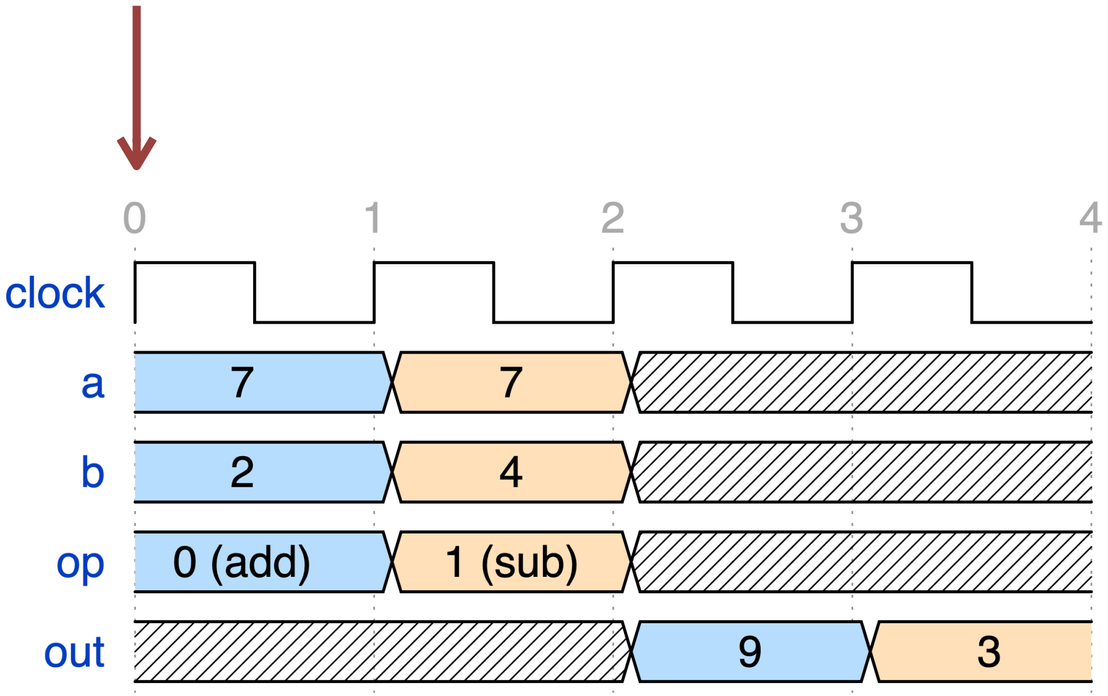

```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a; DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

```
prot sub<DUT: ALU>(a, b, out) {
  DUT.a := a; DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0



add(a=?, b=?, out=?)

Constraints: ∅

sub(a=?, b=?, out=?)

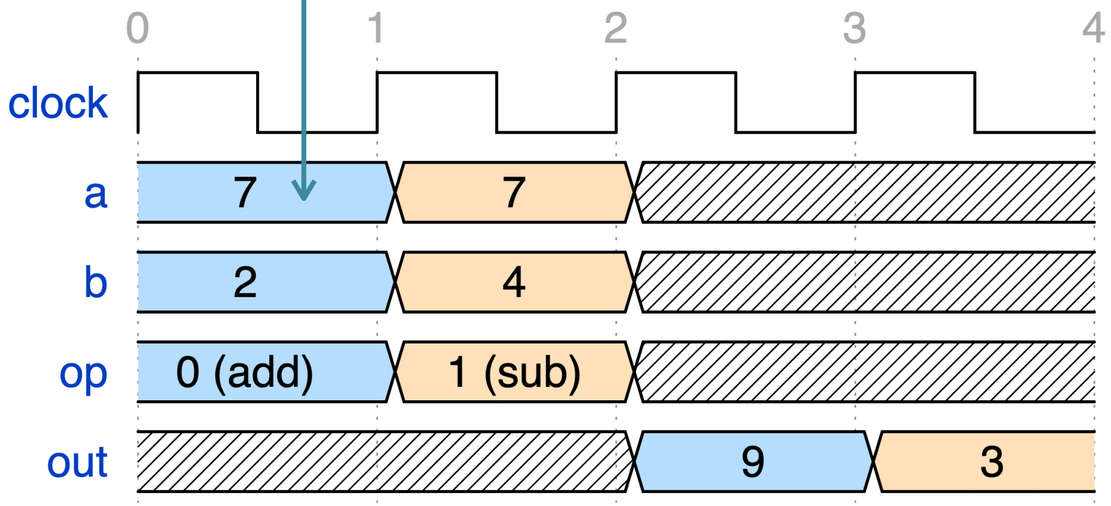
Constraints: ∅

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: a = 7



add(a=?, b=?, out=?)

Constraints: ∅

sub(a=?, b=?, out=?)

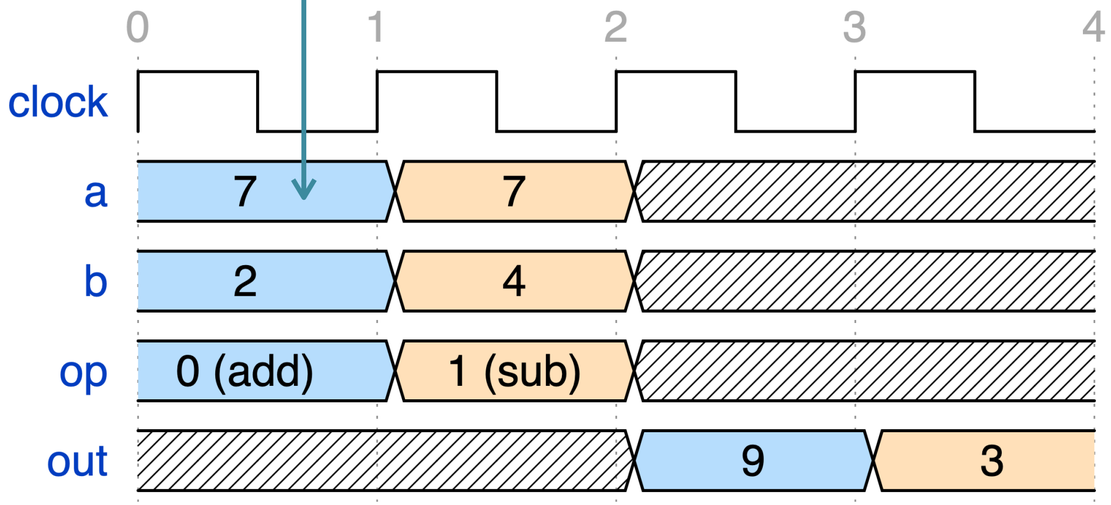
Constraints: ∅

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: a = 7



add(a=7, b=?, out=?)

Constraints: { a = 7 }

New!

sub(a=?, b=?, out=?)

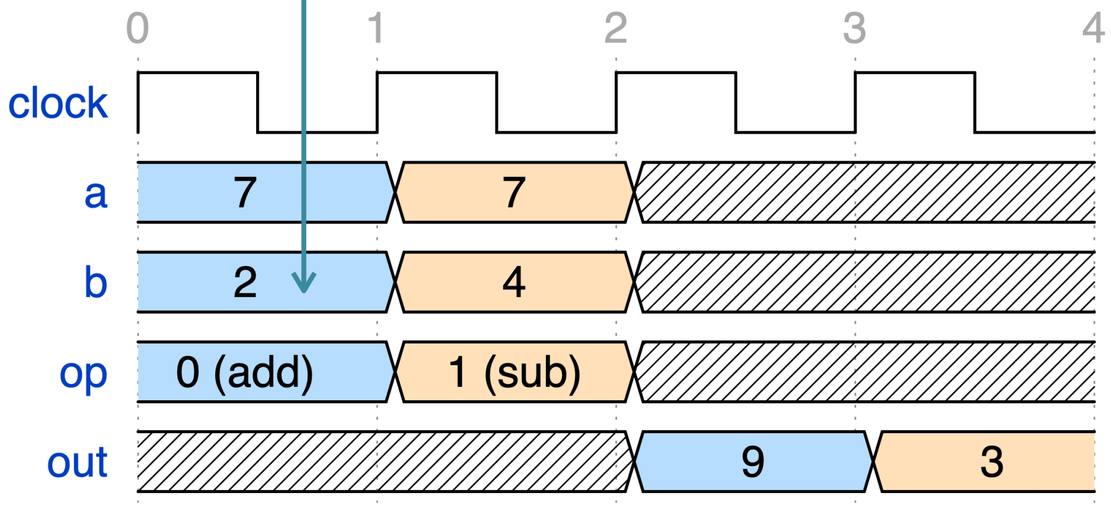
Constraints: ∅

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: `b = 2`



`add(a=7, b=?, out=?)`

Constraints: `{ a = 7 }`

`sub(a=?, b=?, out=?)`

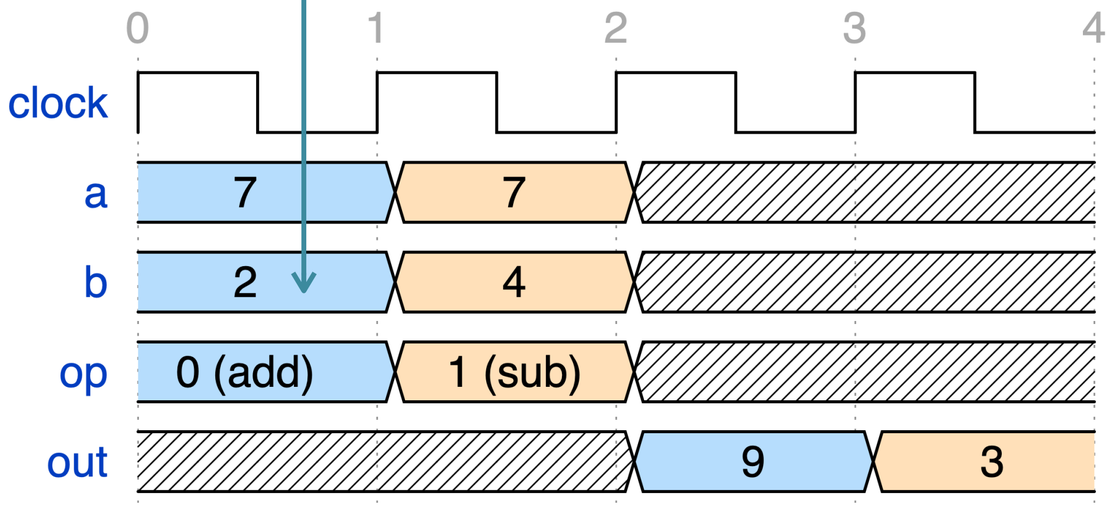
Constraints: ∅

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: `b = 2`



`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2 }`

New!

`sub(a=?, b=?, out=?)`

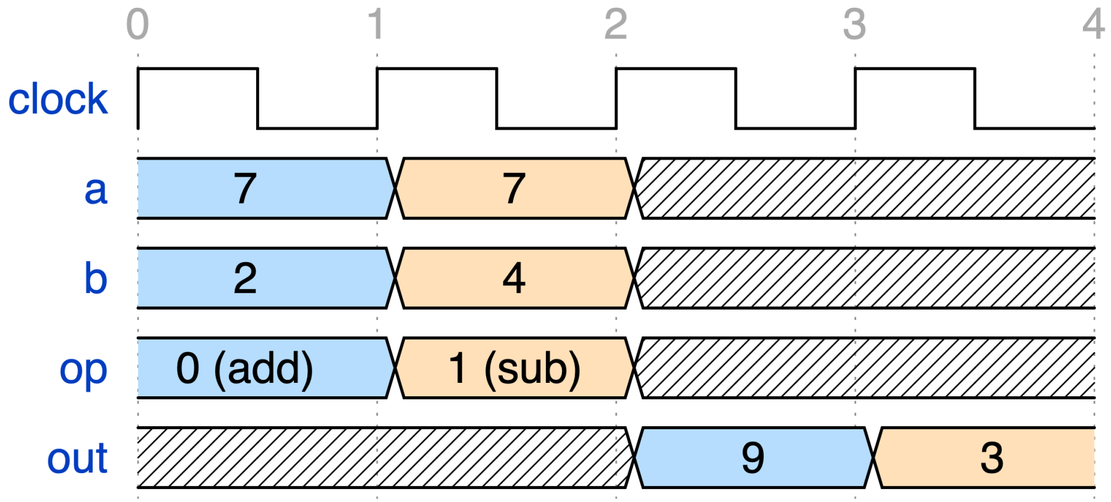
Constraints: ∅

Check if current waveform value of `op =? 0`

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0



`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2 }`

`sub(a=?, b=?, out=?)`

Constraints: ∅

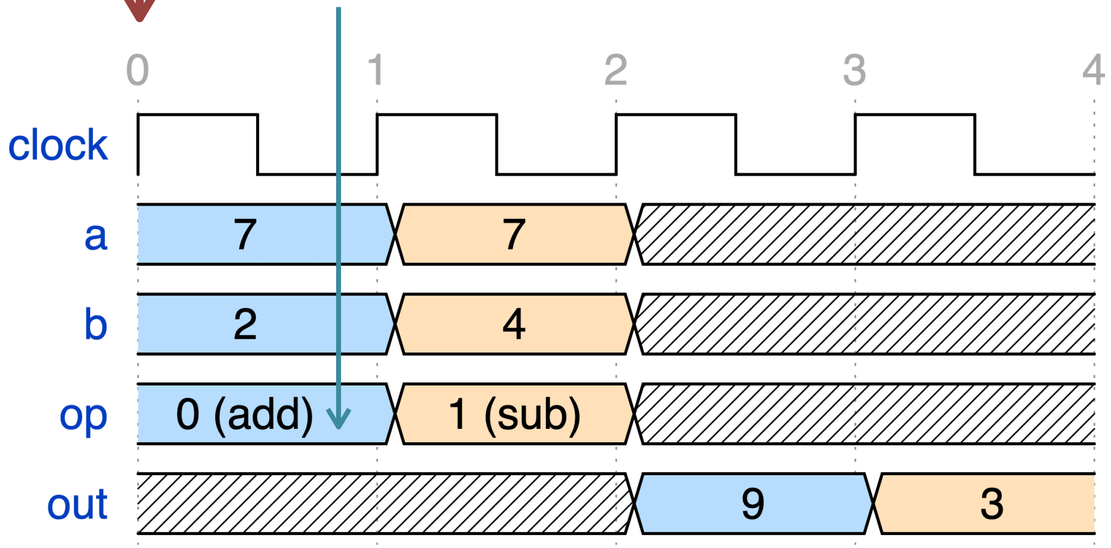
Check if current waveform value of `op =? 0`

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

`op = 0` at current clock cycle ✓



`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2 }`

`sub(a=?, b=?, out=?)`

Constraints: ∅

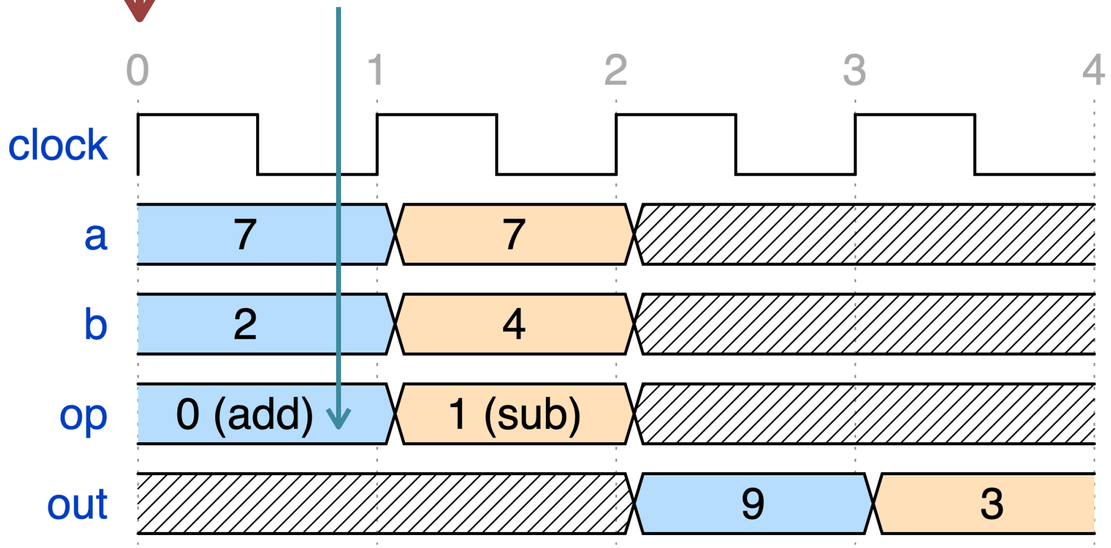
Check if current waveform value of `op =? 0`

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

`op = 0` at current clock cycle ✓



`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2, 0 = 0 }`

New!

`sub(a=?, b=?, out=?)`

Constraints: ∅

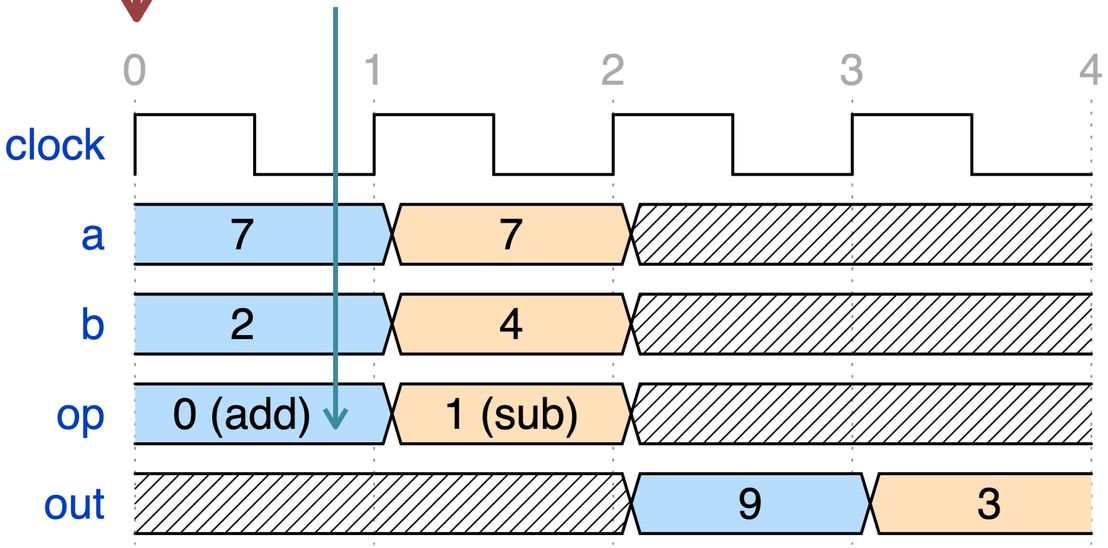
Check if current waveform value of `op =? 0`

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

`op = 0` at current clock cycle ✓



Waveform value of **op** @ current cycle

`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2, 0 = 0 }`

RHS of `DUT.op := 0`

`sub(a=?, b=?, out=?)`

Constraints: ∅

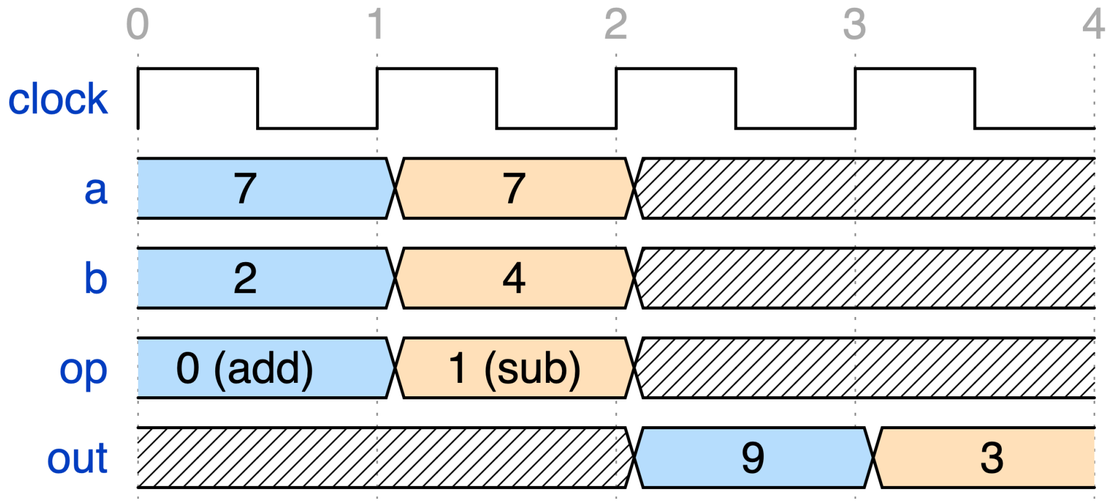
Cycle 0

Execution paths are synchronized when `step()` is called:

Suspend checking **add**, switch to checking **sub**

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```



`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2, 0 = 0 }`

`sub(a=?, b=?, out=?)`

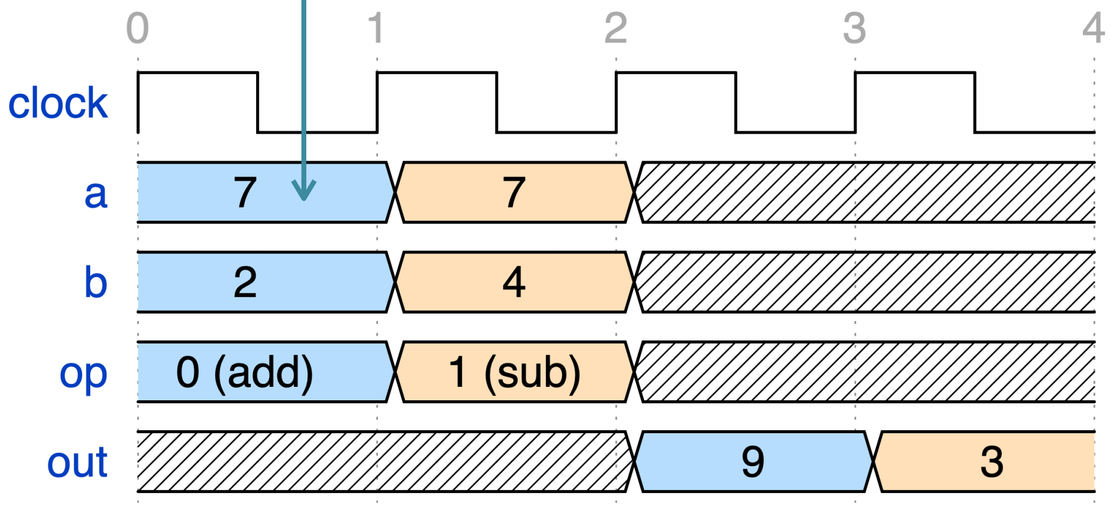
Constraints: ∅

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: a = 7



add(a=7, b=2, out=?)

Constraints: { a = 7, b = 2, 0 = 0 }

sub(a=7, b=?, out=?)

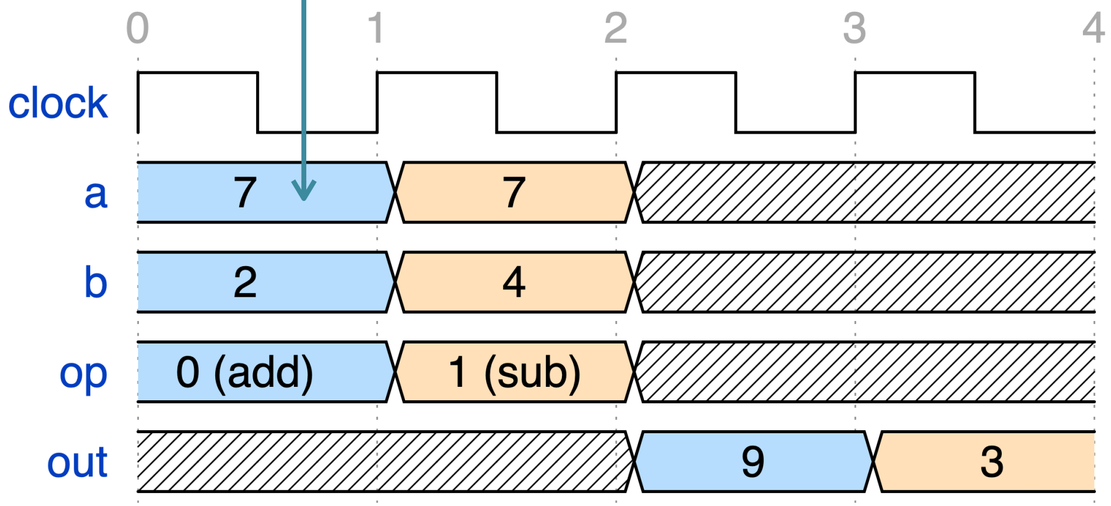
Constraints: ∅

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: a = 7



add(a=7, b=2, out=?)

Constraints: { a = 7, b = 2, 0 = 0 }

sub(a=7, b=?, out=?)

Constraints: { a = 7 }

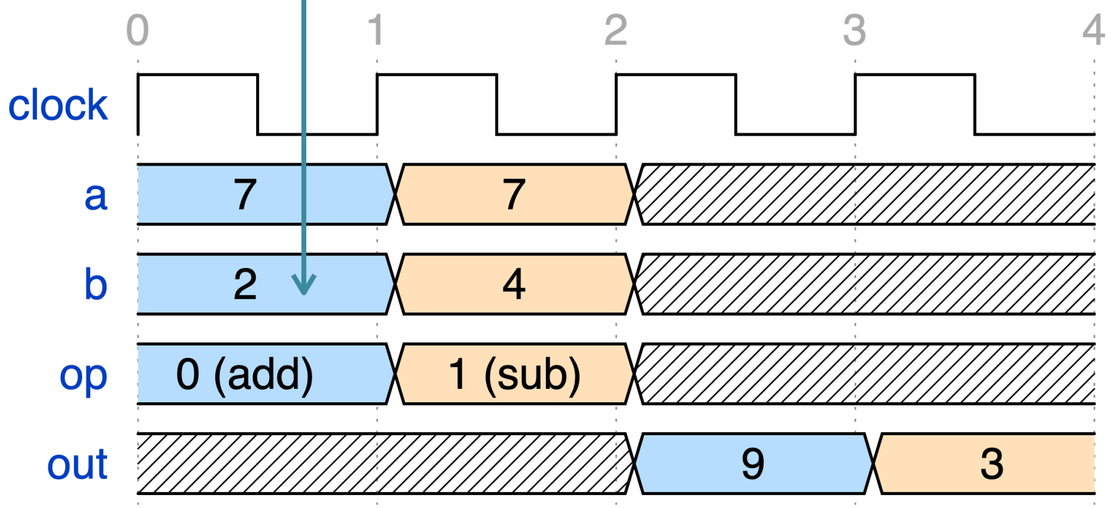
New!

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: b = 2



add(a=7, b=2, out=?)

Constraints: { a = 7, b = 2, 0 = 0 }

sub(a=7, b=?, out=?)

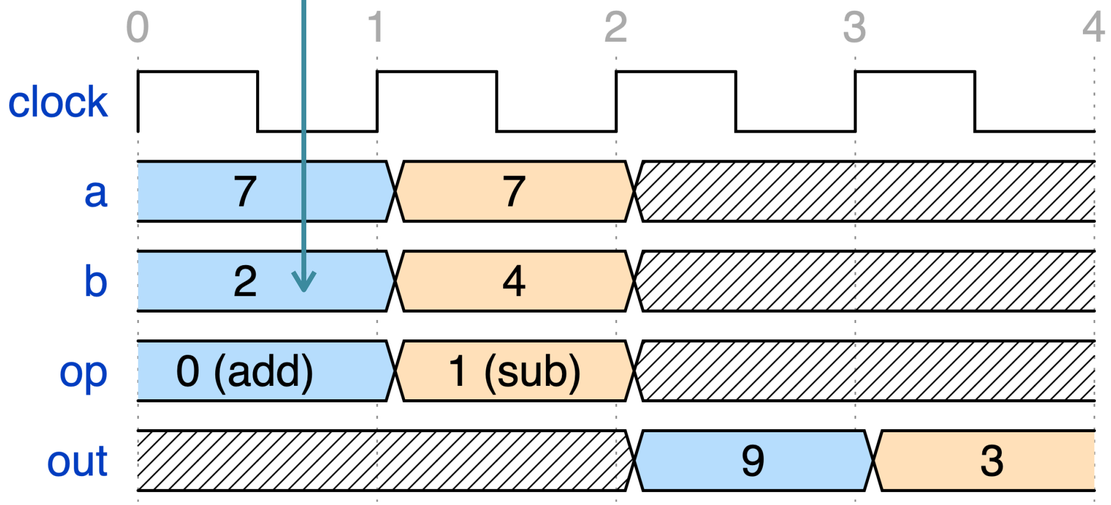
Constraints: { a = 7 }

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

At current clock cycle: `b = 2`



`add(a=7, b=2, out=?)`

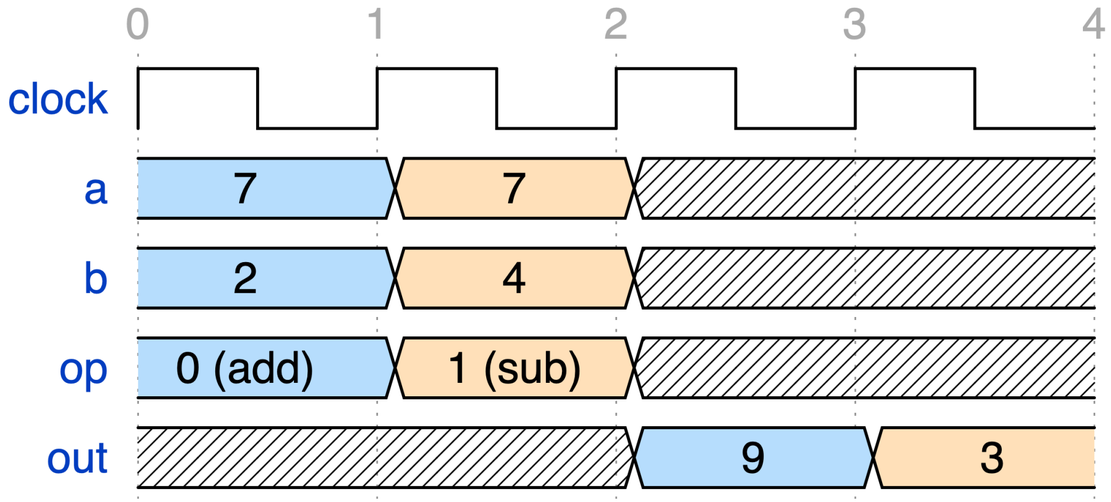
Constraints: `{ a = 7, b = 2, 0 = 0 }`

`sub(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2 }`

New!

Cycle 0



Check if current waveform value of op =? 1

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

add(a=7, b=2, out=?)

Constraints: { a = 7, b = 2, 0 = 0 }

sub(a=7, b=2, out=?)

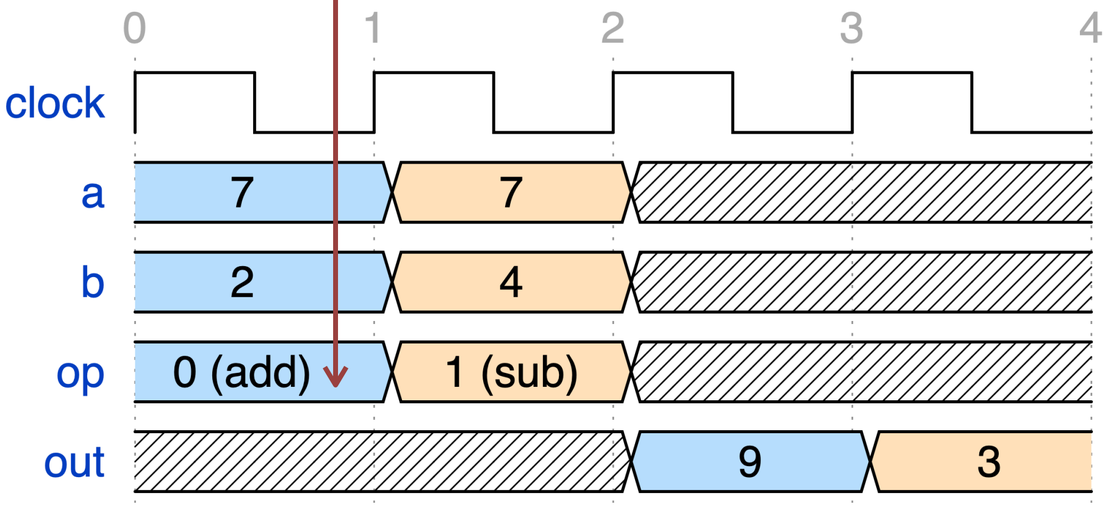
Constraints: { a = 7, b = 2 }

Check if current waveform value of `op =? 1`

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Cycle 0

`op = 0` at current clock cycle, inconsistent with `DUT.op := 1`



`add(a=7, b=2, out=?)`

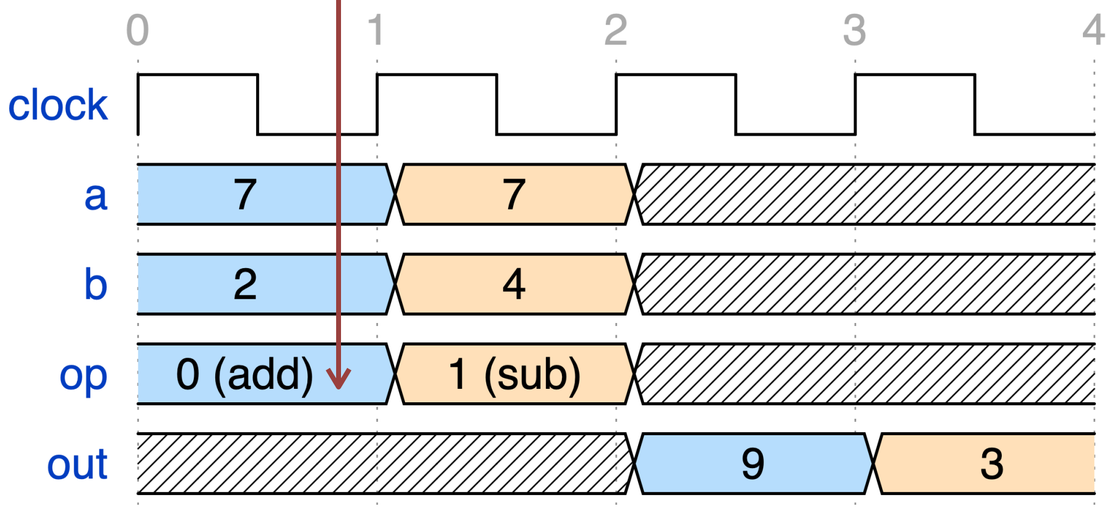
Constraints: `{ a = 7, b = 2, 0 = 0 }`

`sub(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2 }`

Cycle 0

`op = 0` at current clock cycle, inconsistent with `DUT.op := 1`



Check if current waveform value of `op =? 1`

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

`add(a=7, b=2, out=?)`

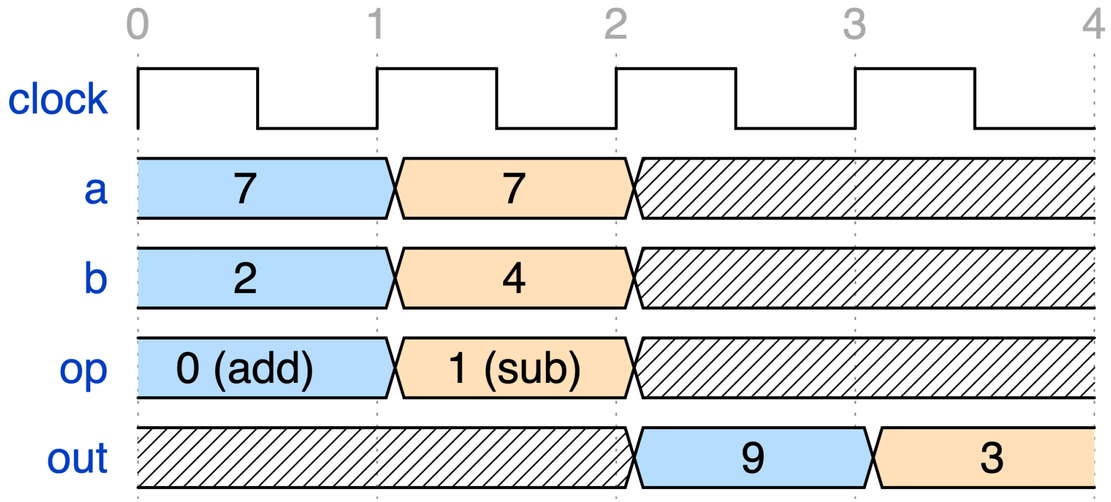
Constraints: `{ a = 7, b = 2, 0 = 0 }`

`sub(a, b, out)`

X Can't have occurred during cycle 0! This path is pruned

Advance to next cycle, prune **sub**

Cycle 1




```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

`add(a=7, b=2, out=?)`

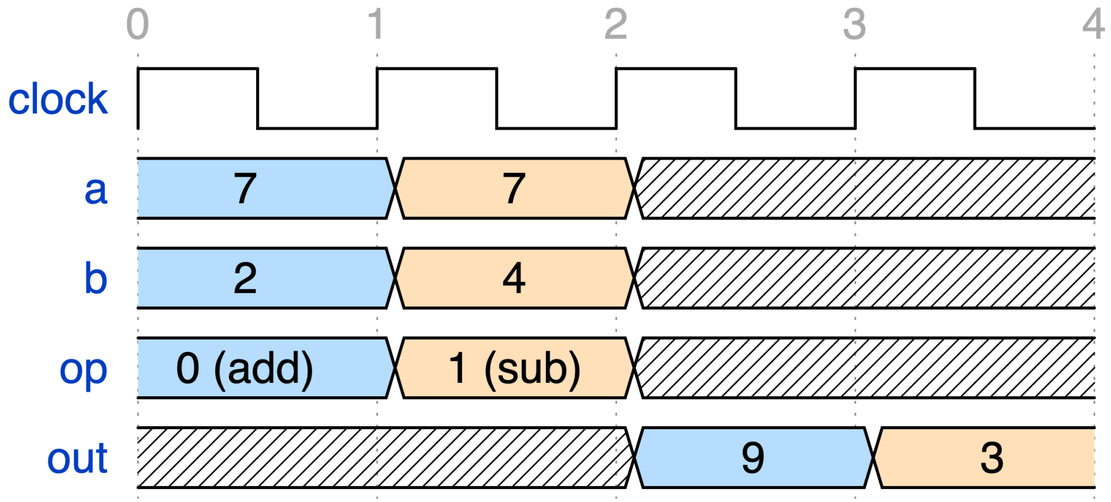
Constraints: `{ a = 7, b = 2, 0 = 0 }`

`sub(a, b, out)`

X Can't have occurred during cycle 0! This path is pruned

Cycle 1

`DontCare` assignments release constraints



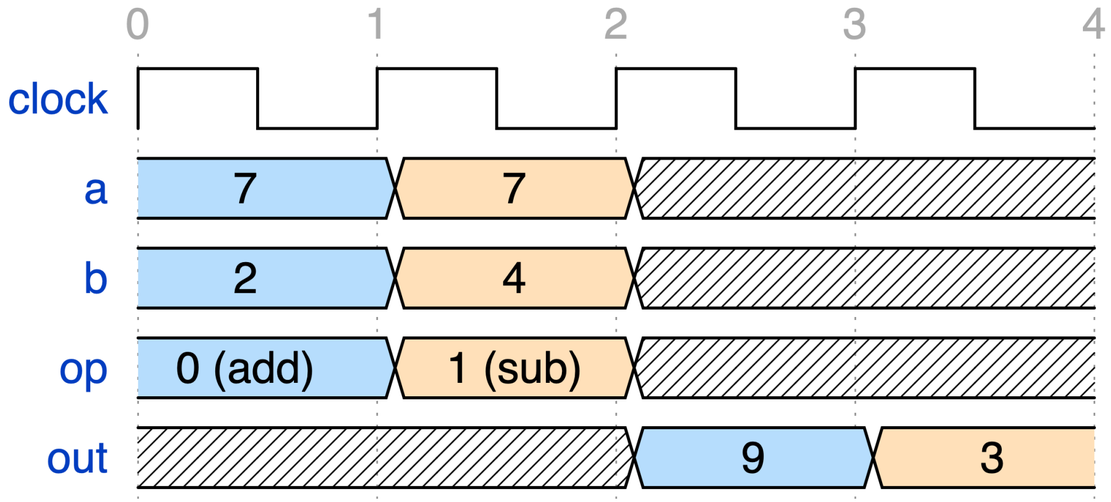

```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

`add(a=7, b=2, out=?)`

Constraints: `{ a = 7, b = 2, 0 = 0 }`

Cycle 1

`DontCare` assignments release constraints



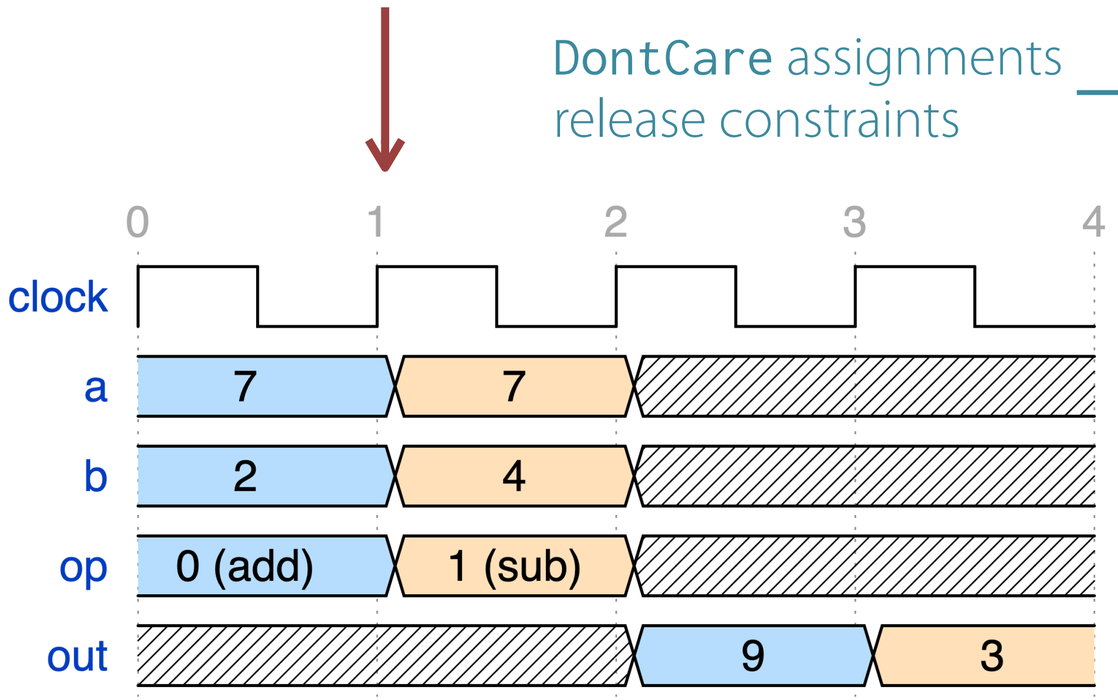

```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

`add(a=7, b=2, out=?)`

Constraints: `{ b = 2, 0 = 0 }`

Constraint `a = 7` removed

Cycle 1



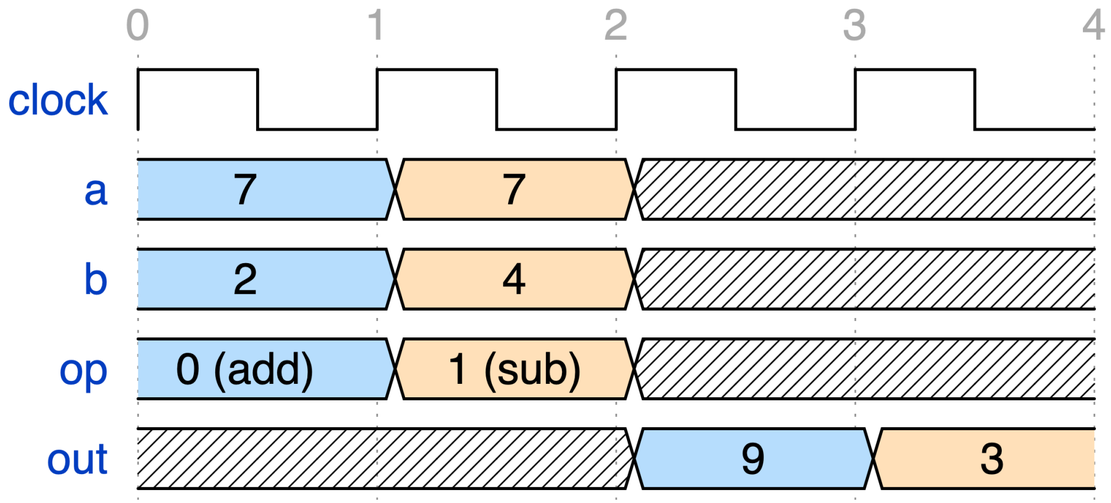
`DontCare` assignments release constraints

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

`add(a=7, b=2, out=?)`

Constraints: `{ b = 2, 0 = 0 }`

Cycle 1



`DontCare` assignments release constraints

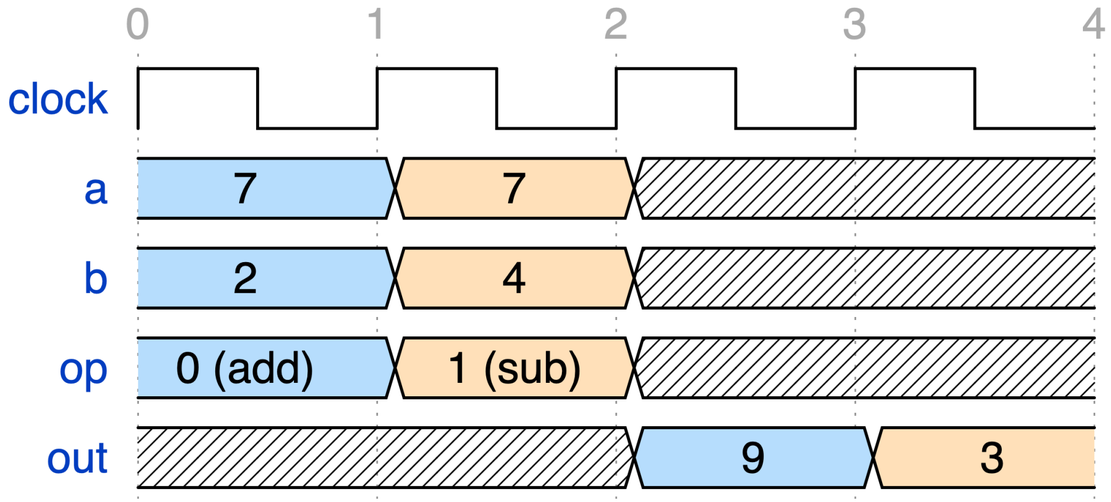
```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

`add(a=7, b=2, out=?)`

Constraints: `{ 0 = 0 }`

Constraint `b = 2` removed

Cycle 1



`DontCare` assignments release constraints

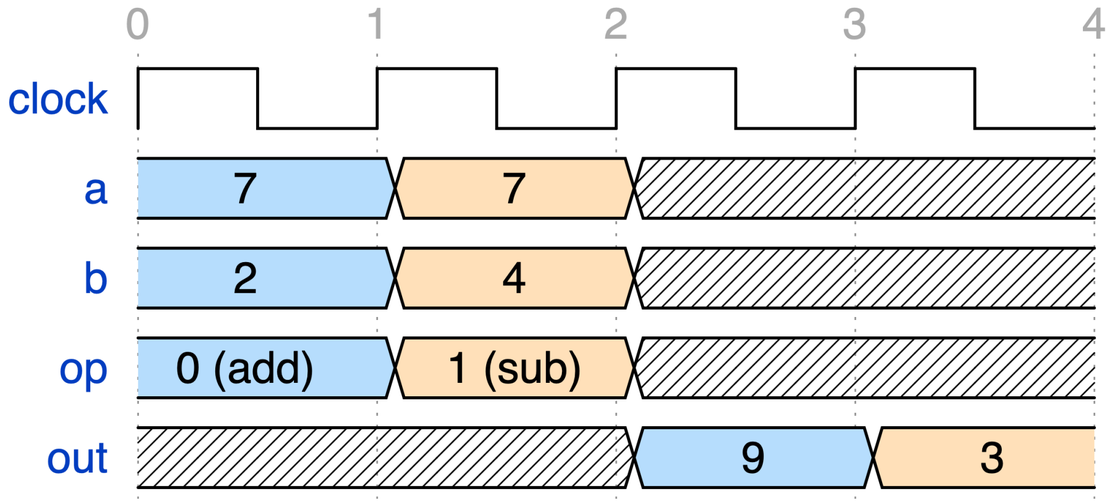
```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

`add(a=7, b=2, out=?)`

Constraints: `{ 0 = 0 }`

Cycle 1

`DontCare` assignments release constraints




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

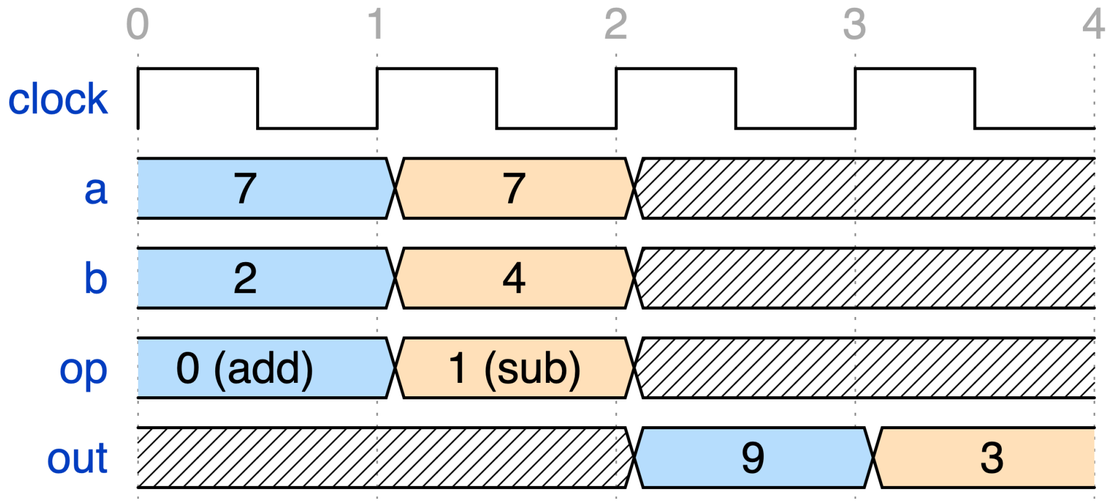
`add(a=7, b=2, out=?)`

Constraints: ∅

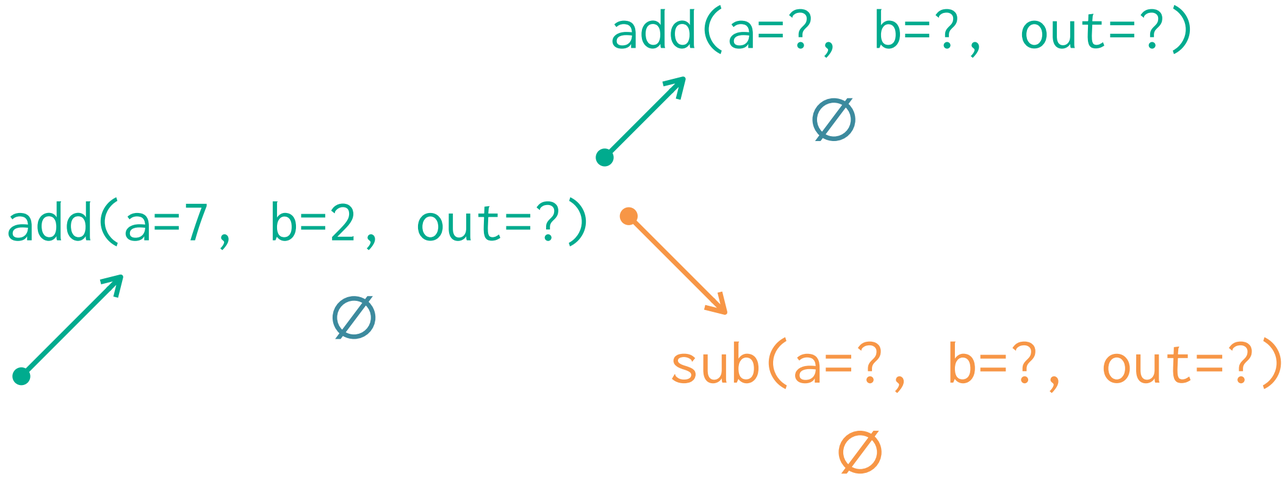
Constraint `0 = 0` removed

Cycle 1

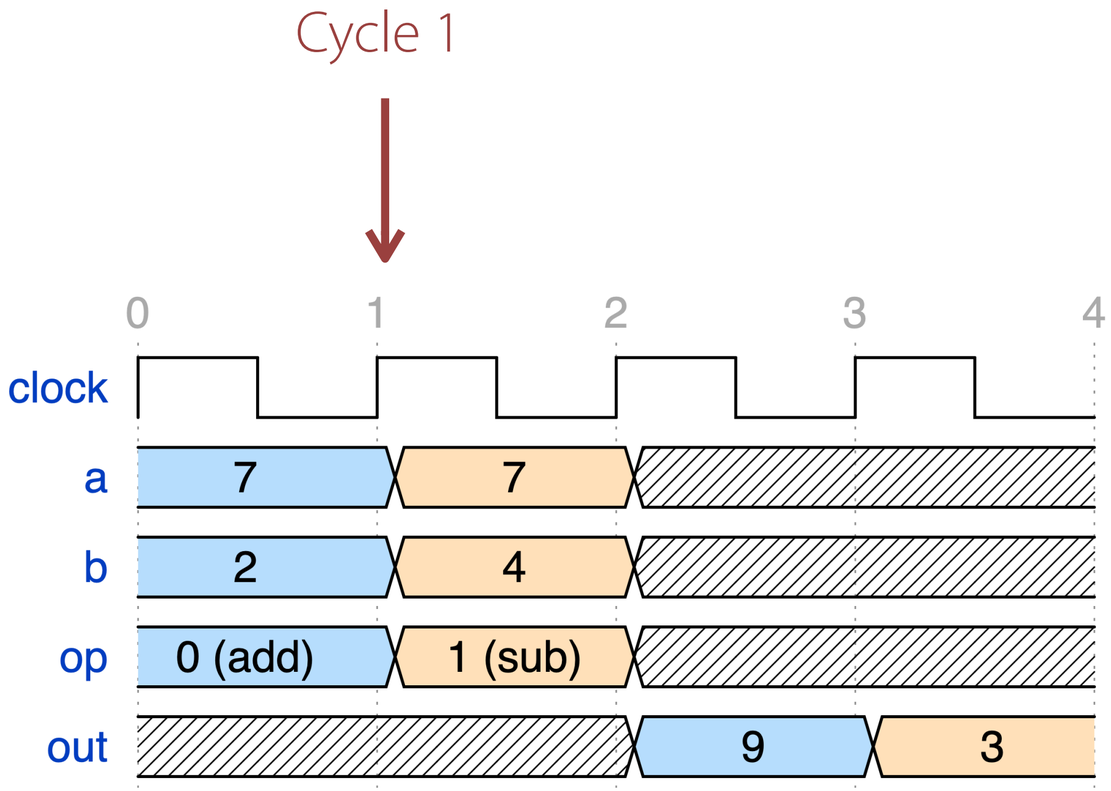
`fork()` spawns one new execution path per protocol




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

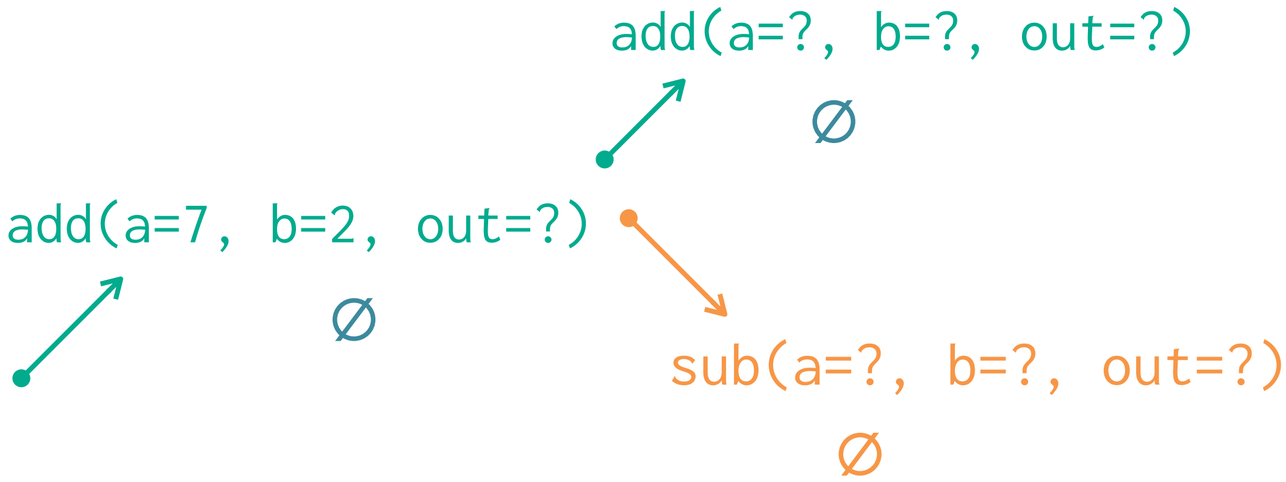


Skipping ahead…

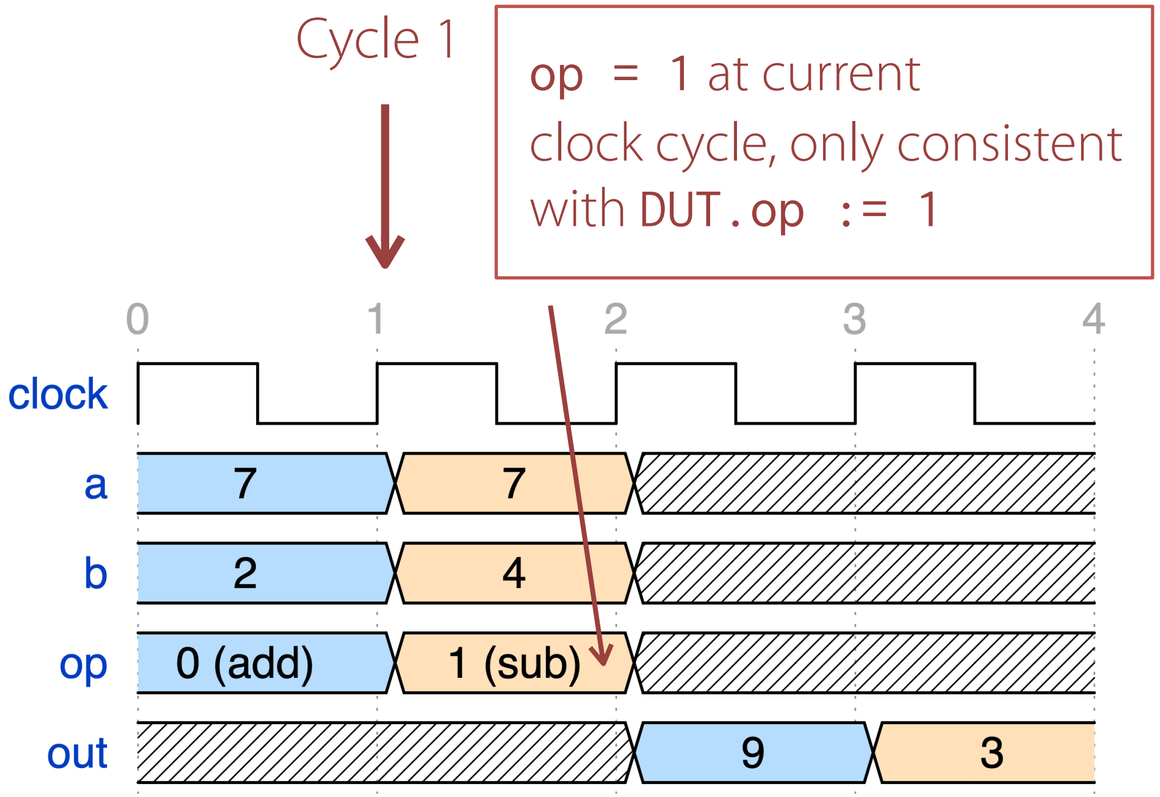



```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

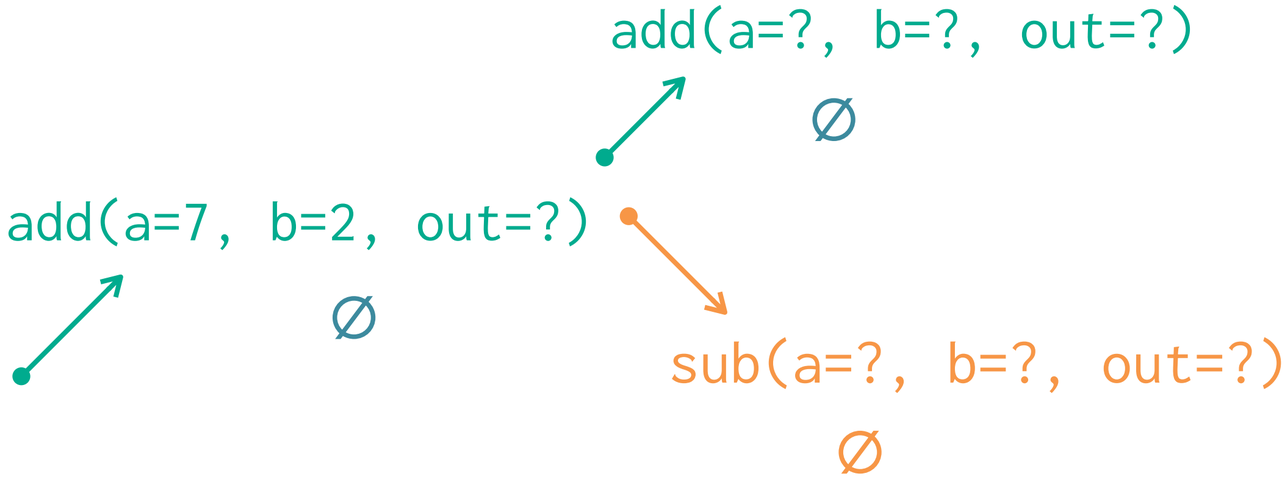


Skipping ahead…

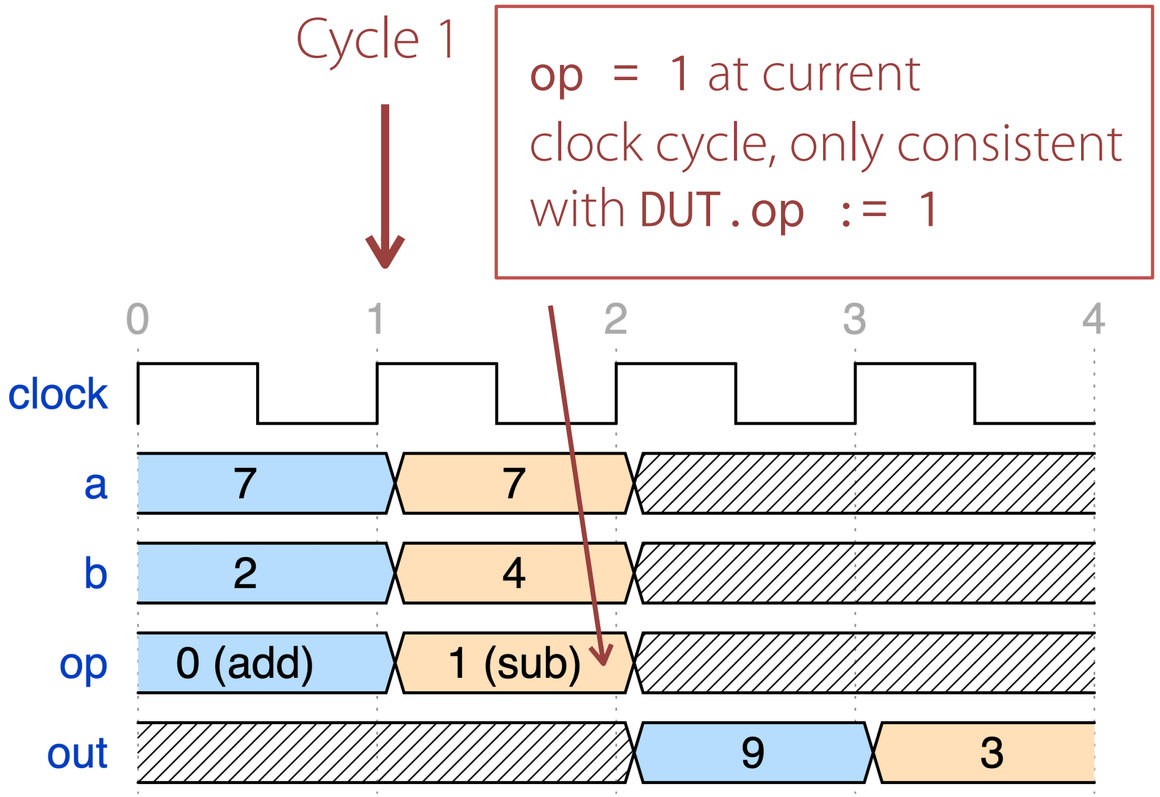



```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```



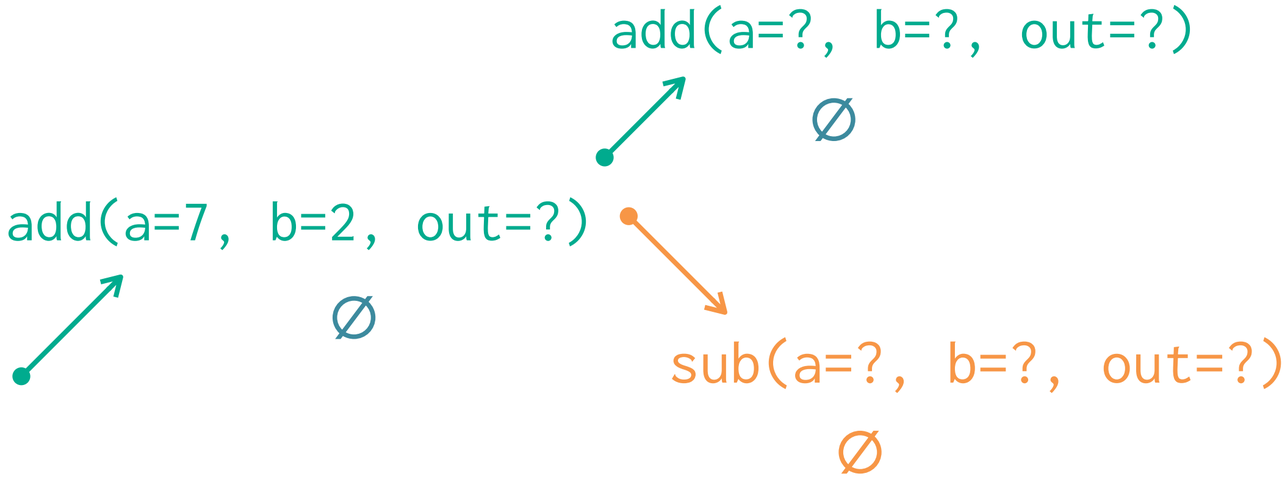
Skipping ahead…



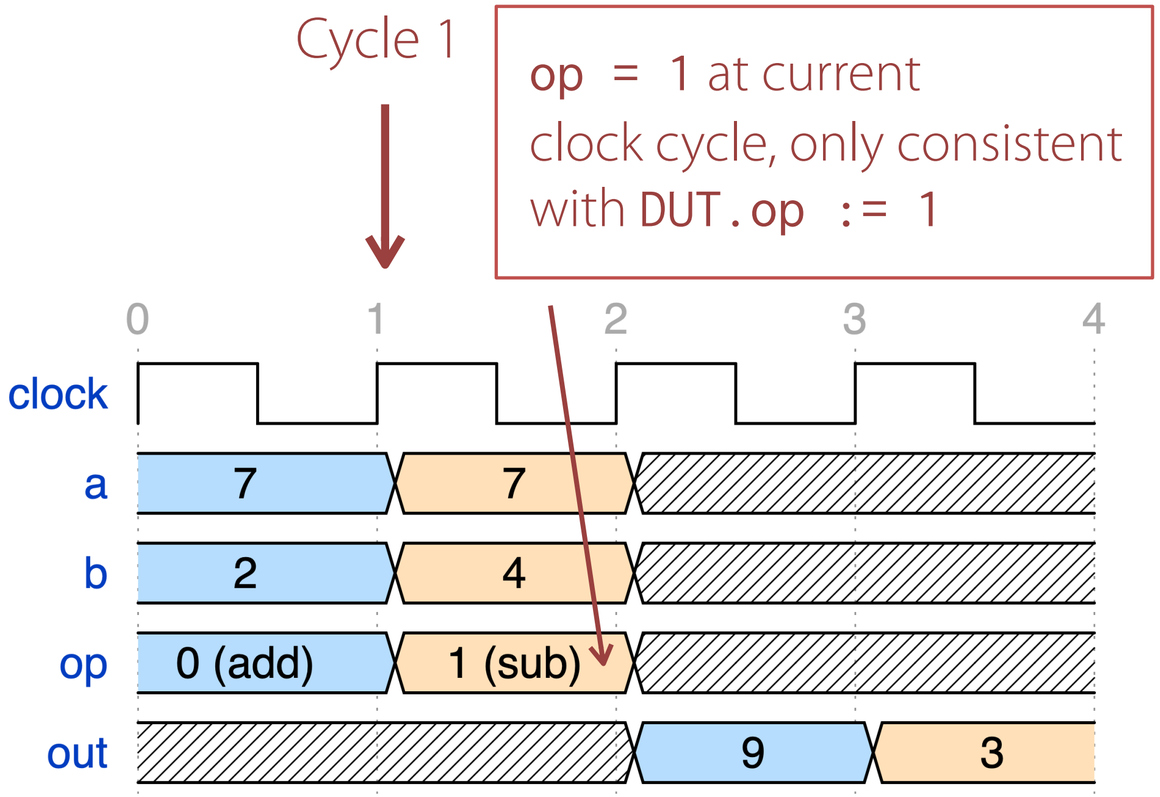

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

Inconsistent!

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```



Skipping ahead…

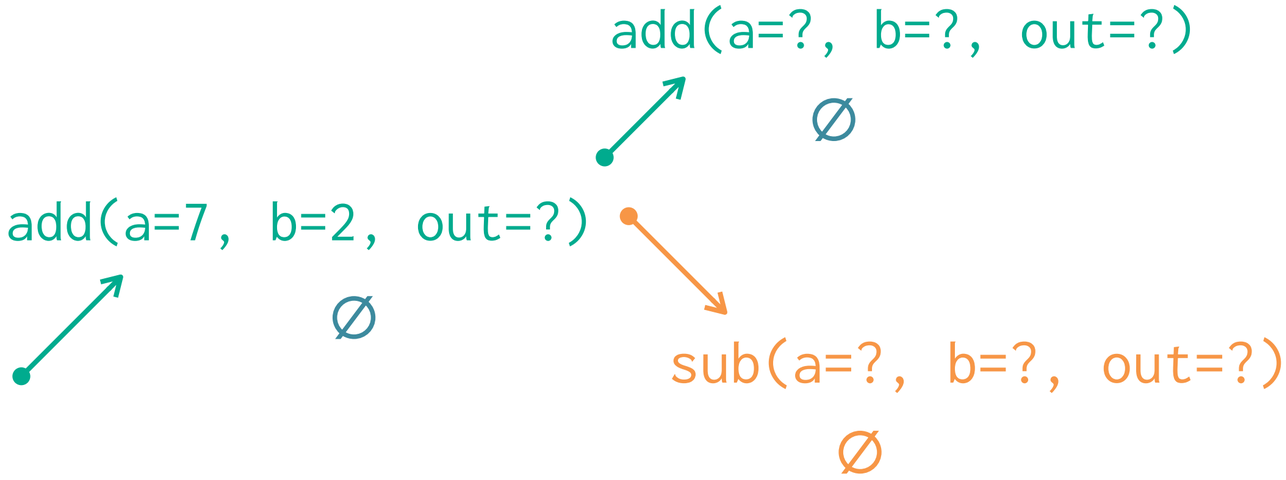



```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

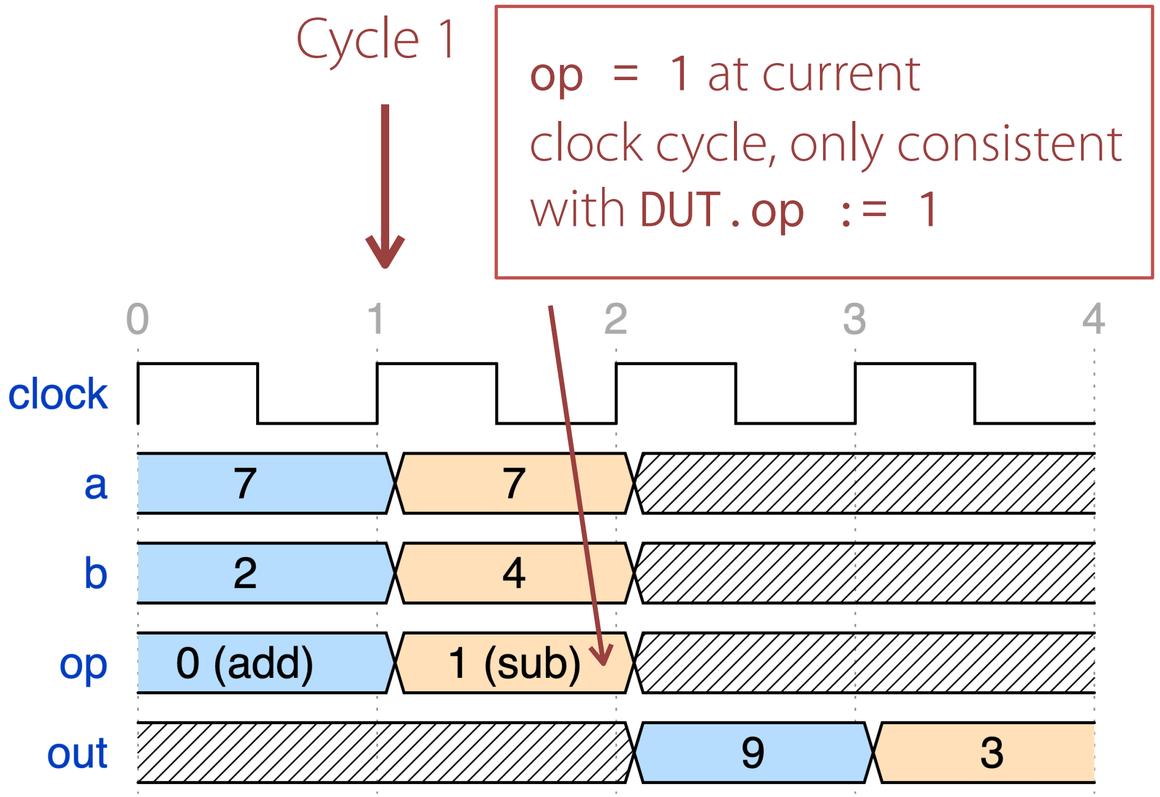
Inconsistent!

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Consistent



Skipping ahead…




```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

Inconsistent!

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Consistent

Can't have occurred during cycle 1! This path is pruned

`add(a=7, b=4, out=?)` **X**

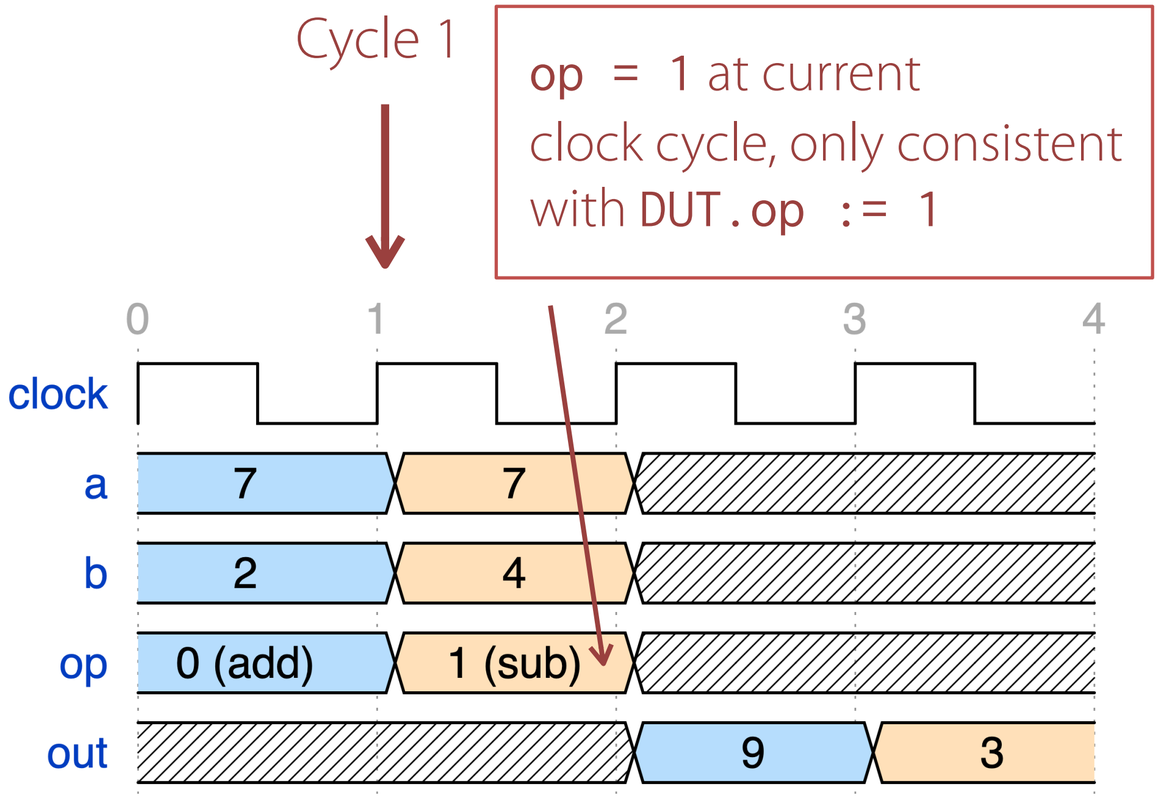
`add(a=7, b=2, out=?)`

∅

`sub(a=?, b=?, out=?)`

∅

Skipping ahead…




```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

Inconsistent!

```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

Consistent

Can't have occurred during cycle 1! This path is pruned

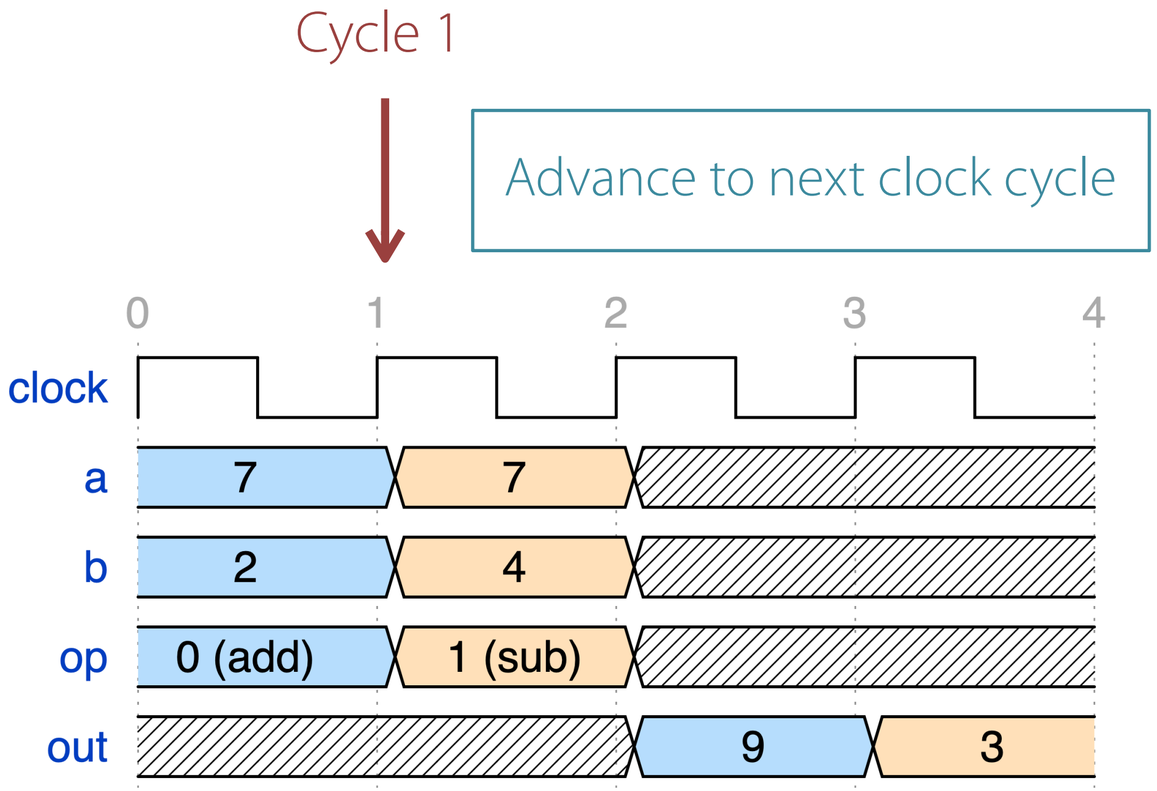
`add(a=7, b=4, out=?)` ✗

`add(a=7, b=2, out=?)` ∅

`sub(a=7, b=4, out=?)` ✓

`{ a = 7, b = 4, 1 = 1 }`

Parent add transaction continues…




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

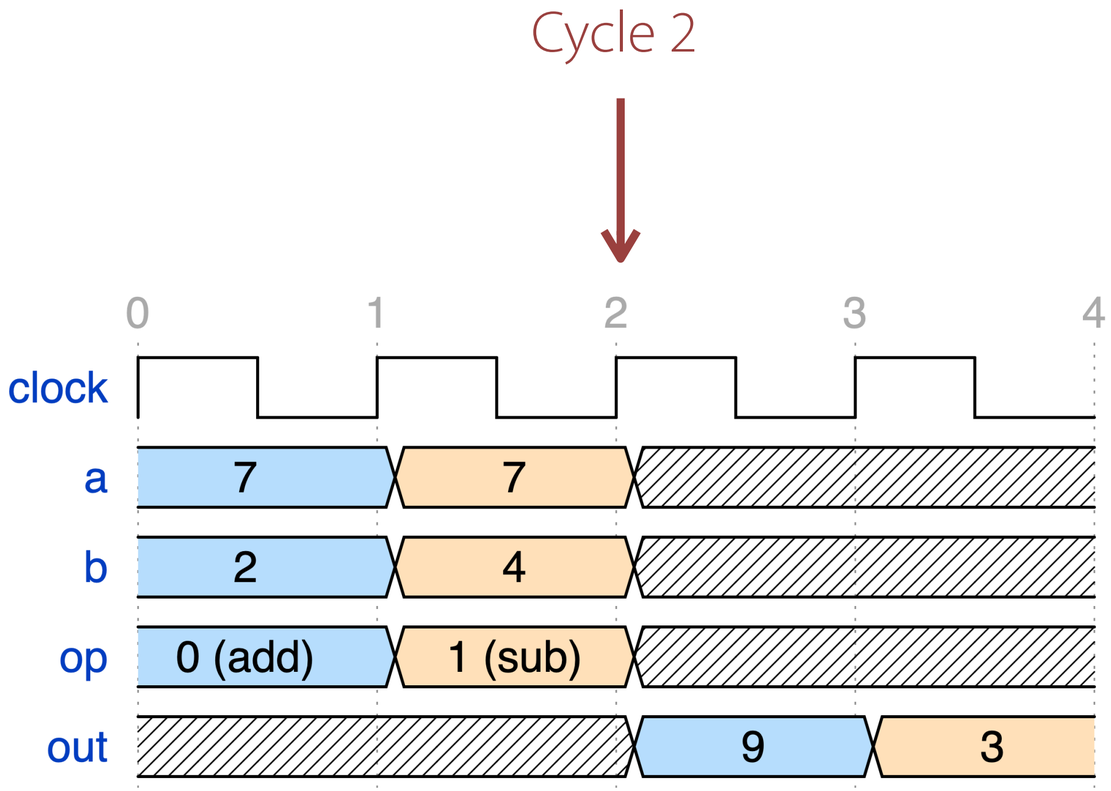
add(a=7, b=2, out=?)

∅

sub(a=7, b=4, out=?)

{ a = 7, b = 4, 1 = 1 }

Parent add transaction continues…



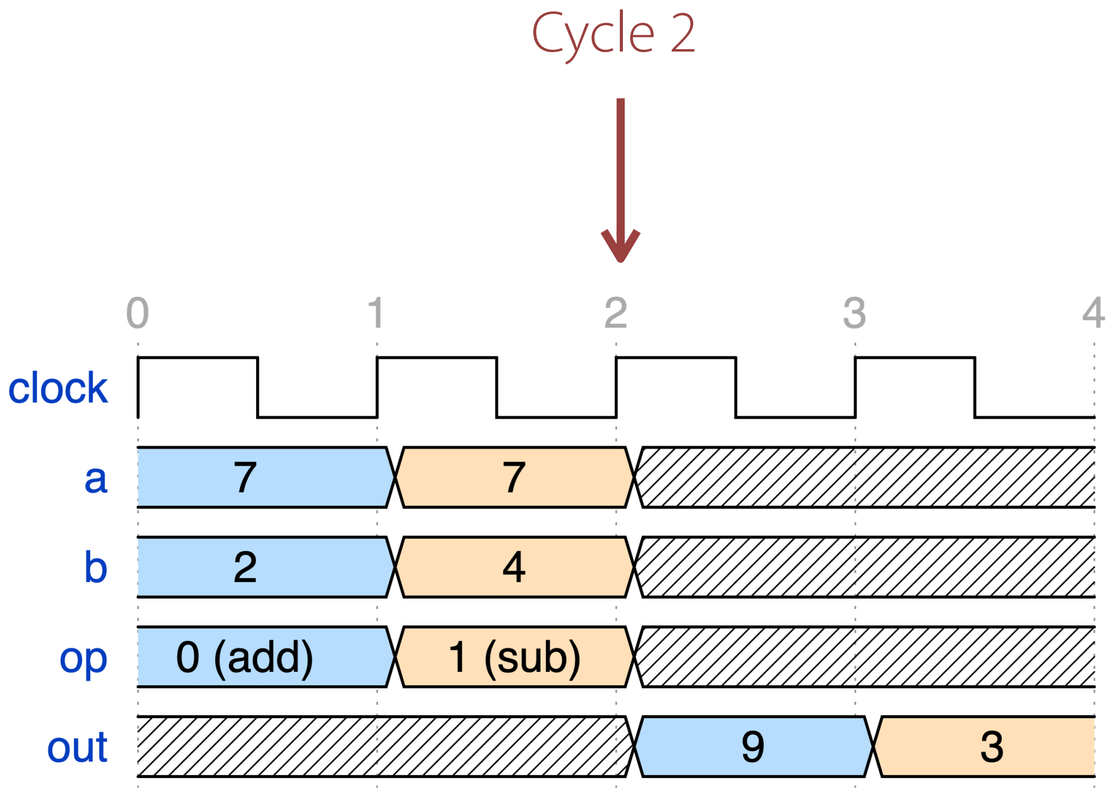

```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

add(a=7, b=2, out=?)

sub(a=7, b=4, out=?)

{ a = 7, b = 4, 1 = 1 }

Parent add transaction continues…




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

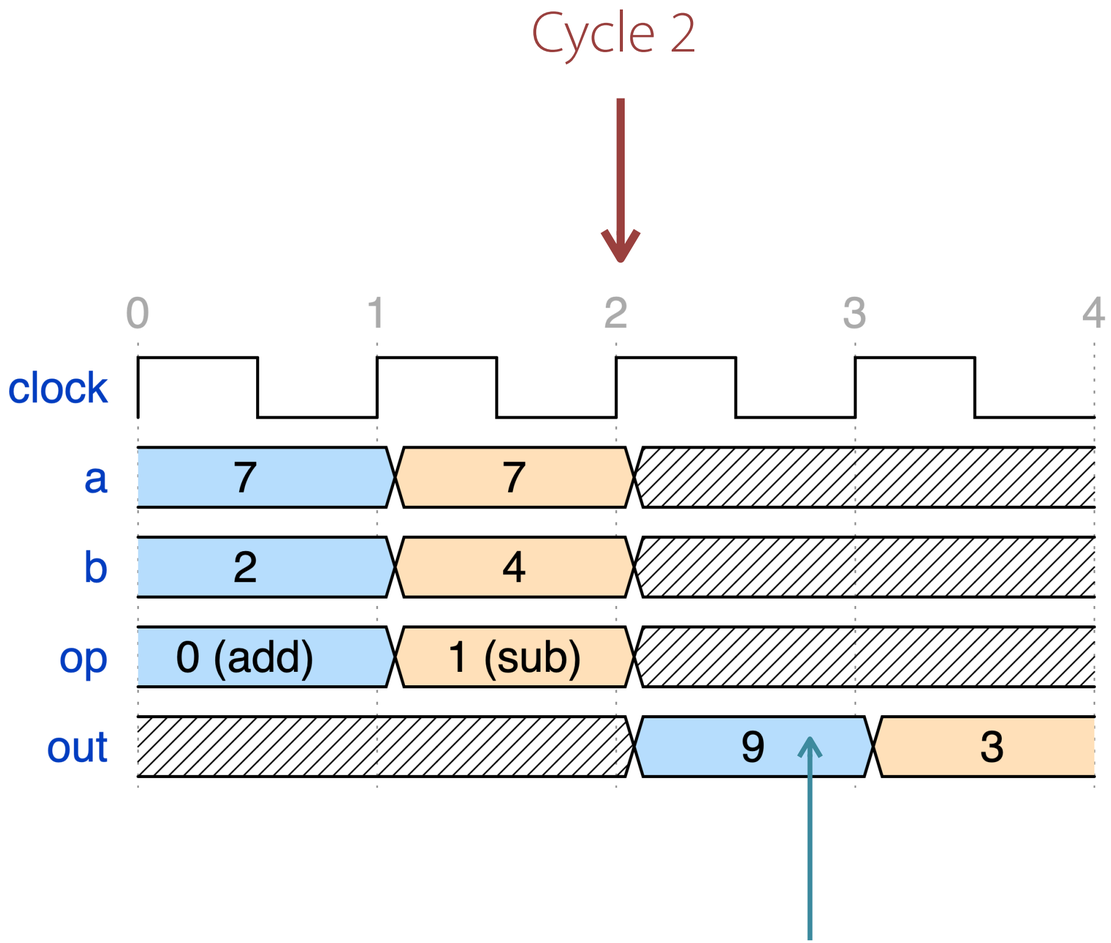
Value of **out** inferred from waveform

add(a=7, b=2, out=?)

sub(a=7, b=4, out=?)

{ a = 7, b = 4, 1 = 1 }

Parent add transaction continues…



At current clock cycle: `out = 9`

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

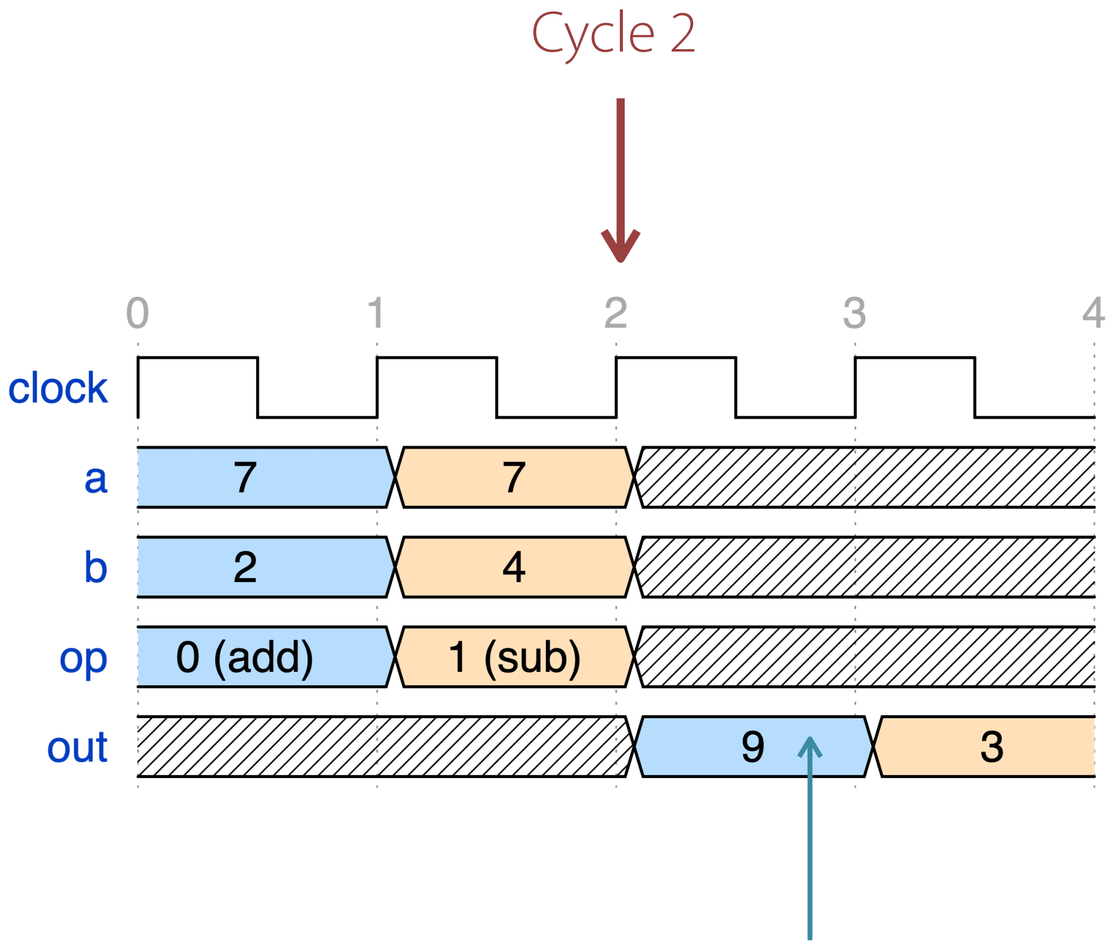
Value of `out` inferred from waveform

`add(a=7, b=2, out=?)`

`sub(a=7, b=4, out=?)`

`{ a = 7, b = 4, 1 = 1 }`

Parent add transaction continues…



At current clock cycle: `out = 9`

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

Value of `out` inferred from waveform

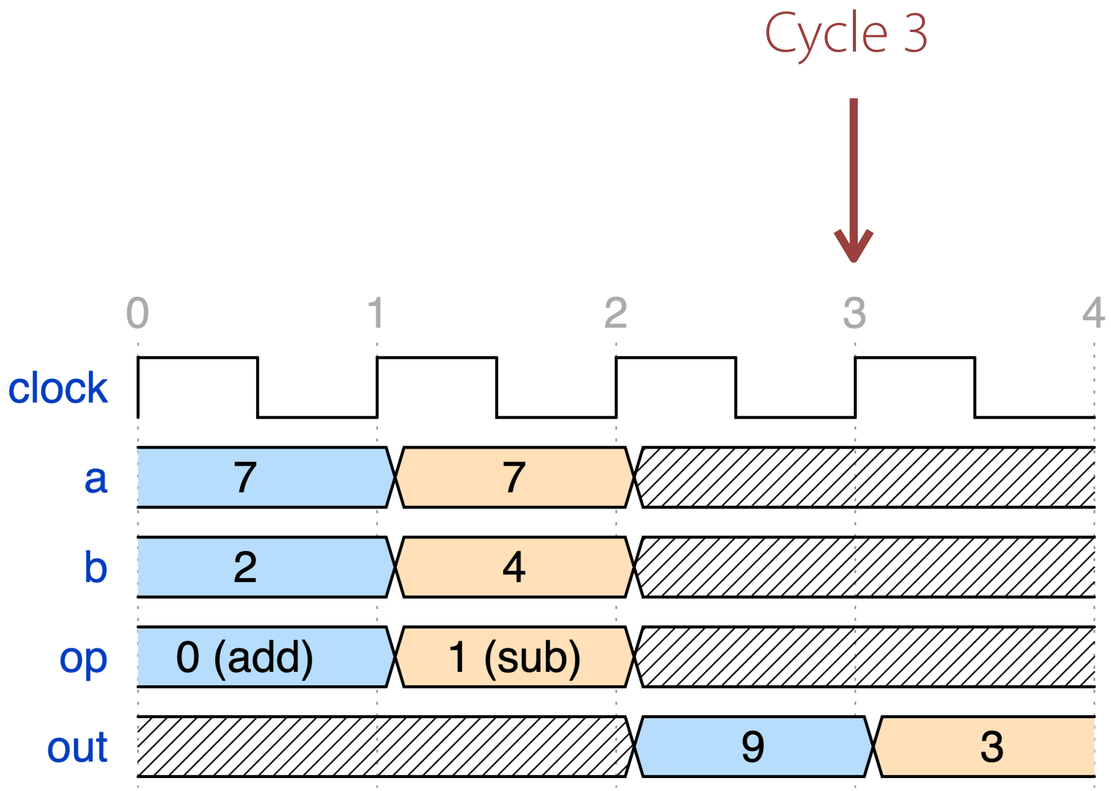
Transaction complete!

`add(a=7, b=2, out=9)`

`sub(a=7, b=4, out=?)`

`{ a = 7, b = 4, 1 = 1 }`

Skipping ahead to end of sub transaction…




```
prot sub<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

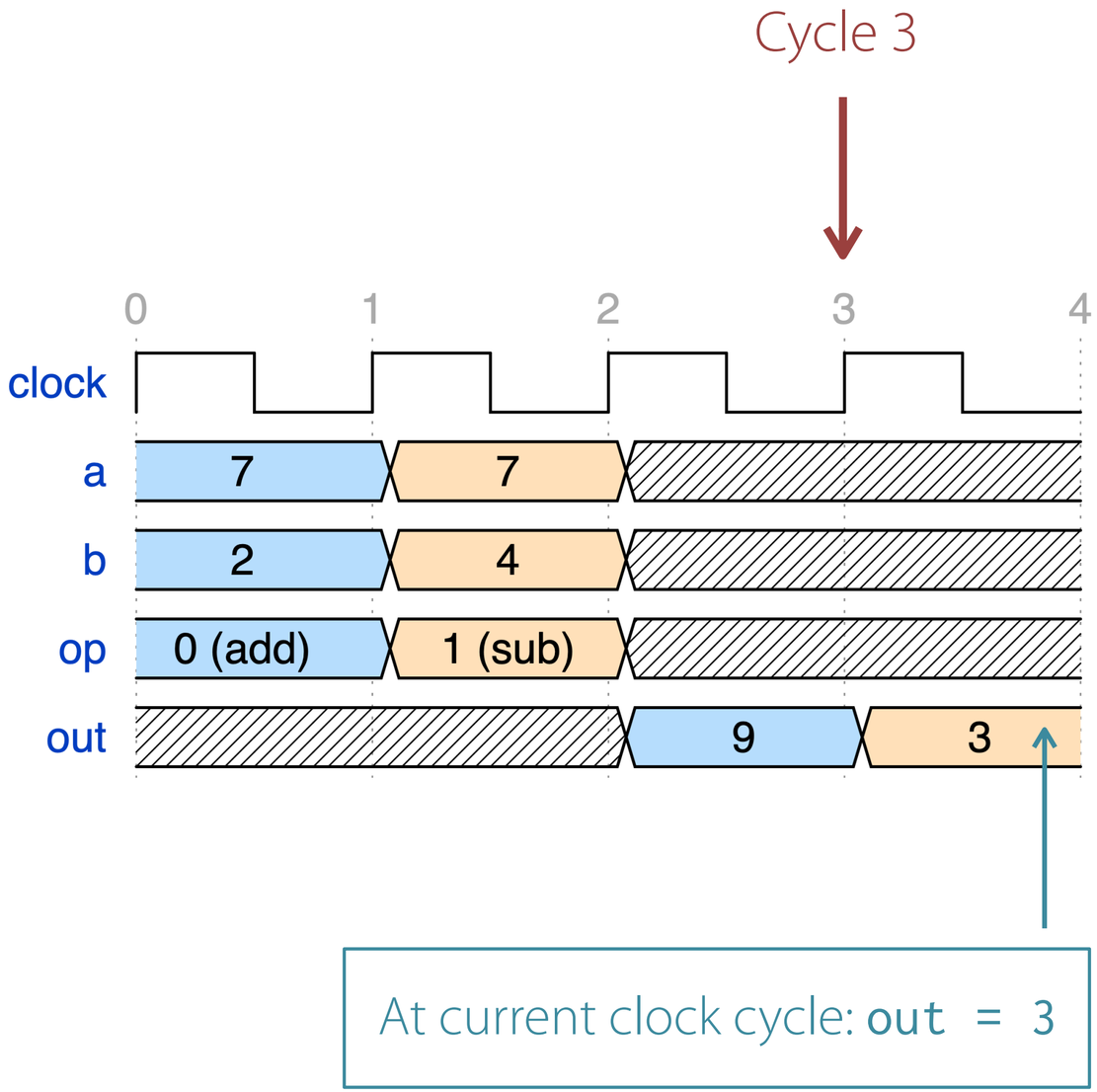
Value of **out** inferred from waveform

Transaction complete!

add(a=7, b=2, out=9)

sub(a=7, b=4, out=?)

Skipping ahead to end of sub transaction…




```
prot sub<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

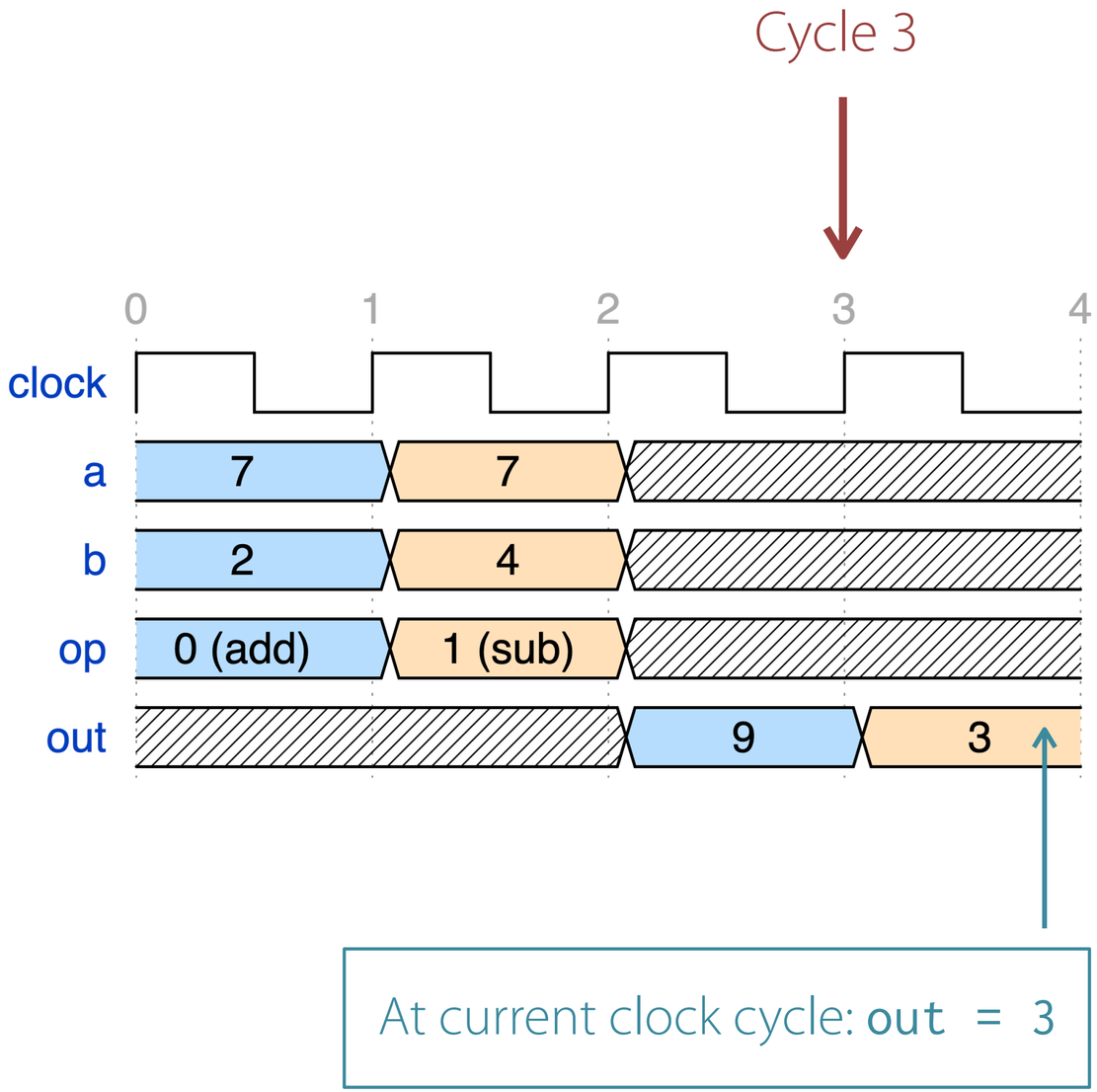
Value of **out** inferred from waveform

Transaction complete!

add(a=7, b=2, out=9)

sub(a=7, b=4, out=?)

Skipping ahead to end of sub transaction…




```
prot sub<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

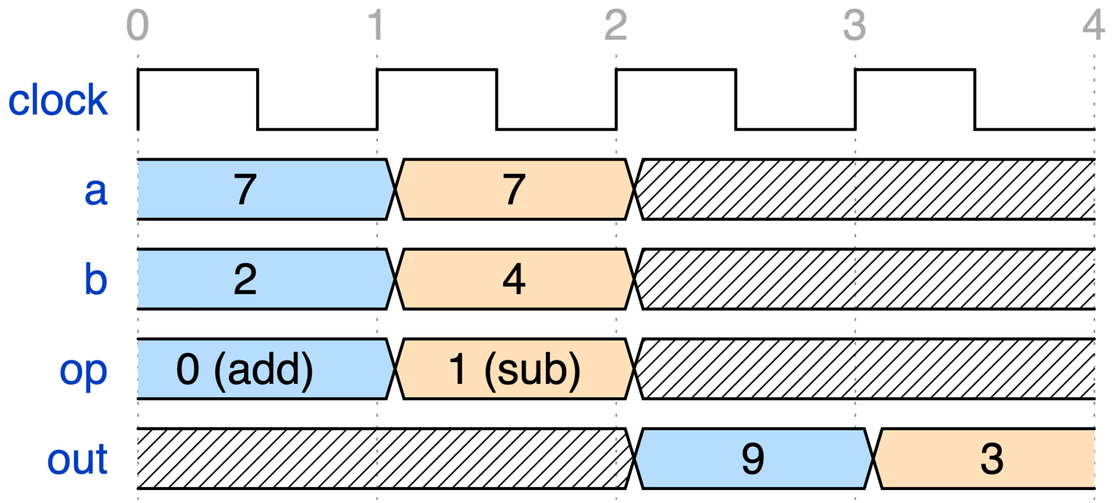
Value of **out** inferred from waveform

Transaction complete!

add(a=7, b=2, out=9)

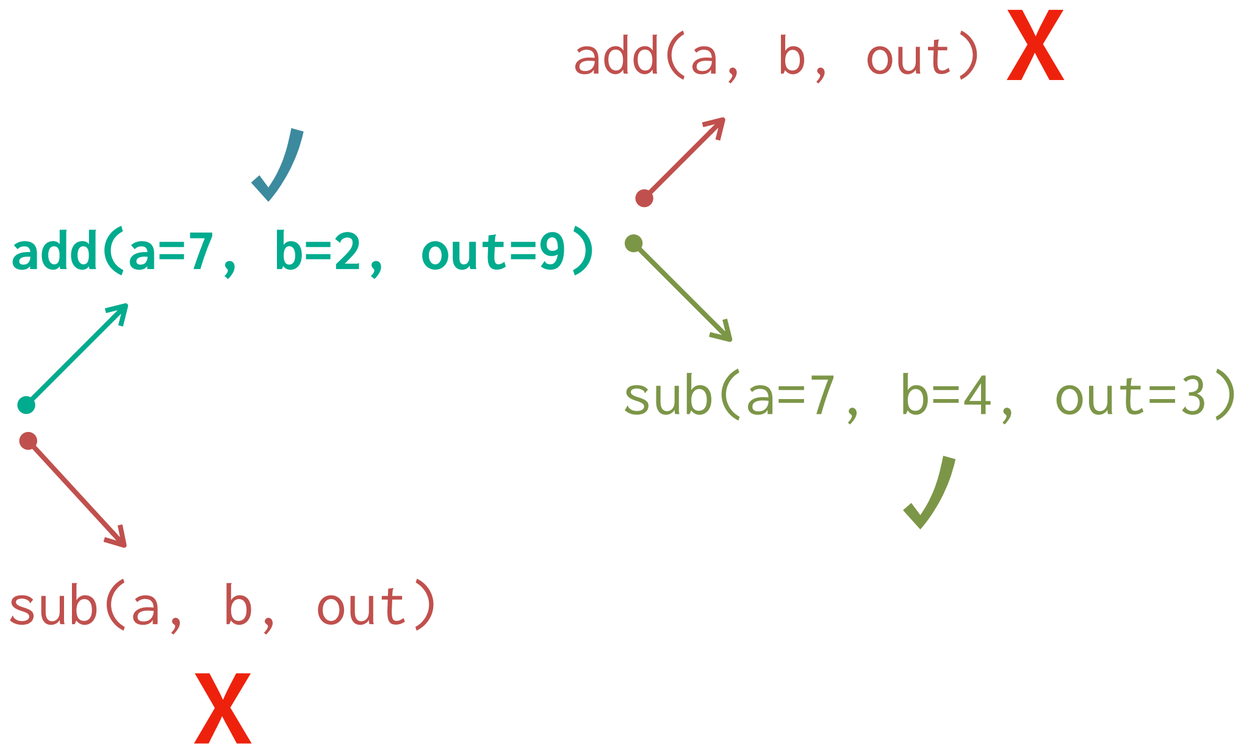
sub(a=7, b=4, out=3)

Transaction complete!



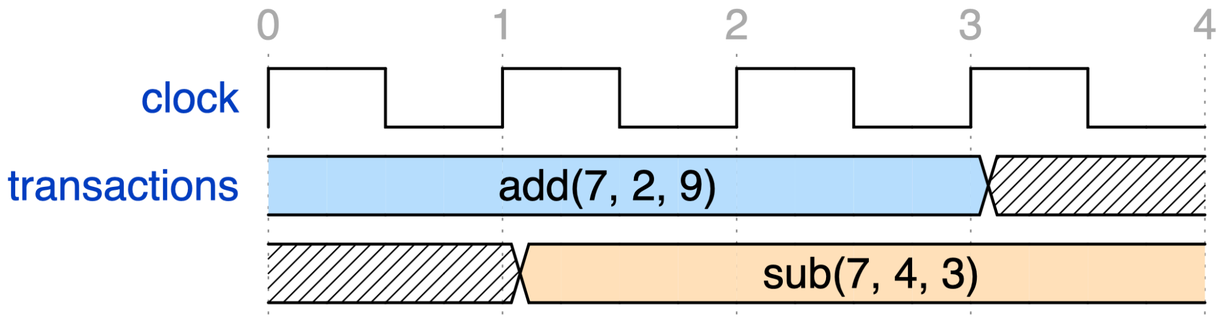
Inferred transaction trace:

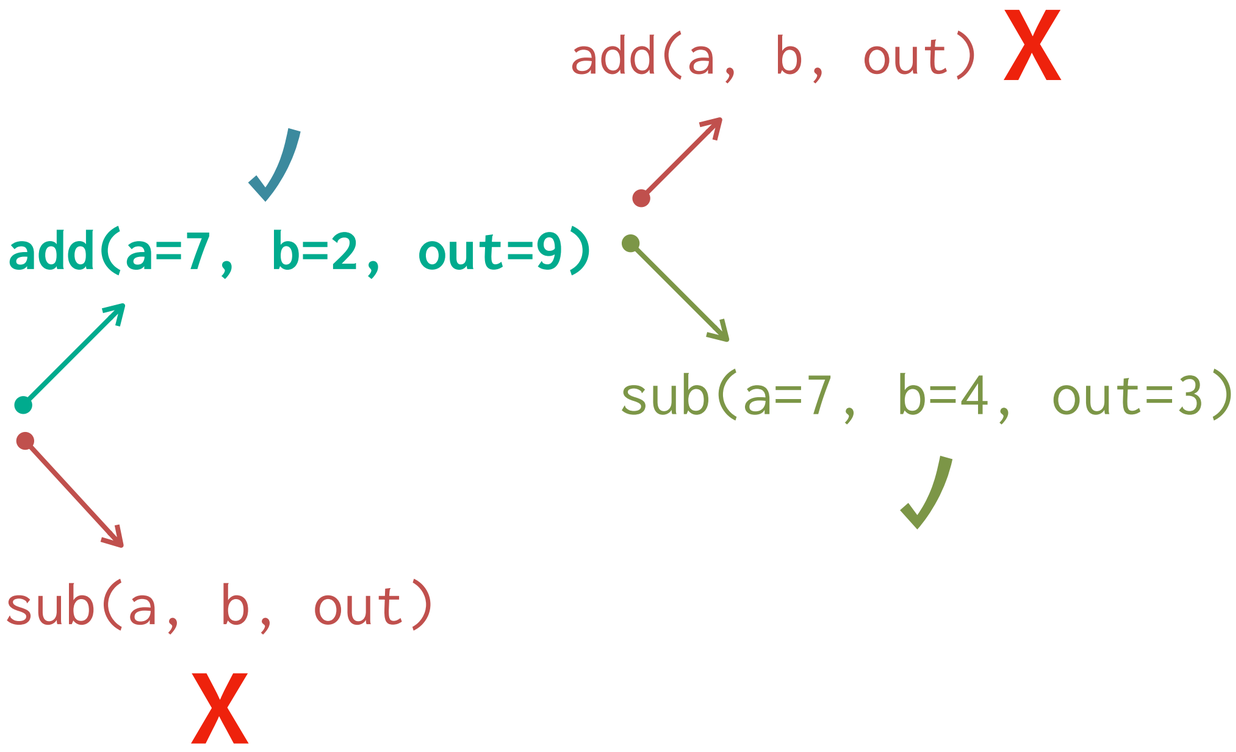
```
add(7, 2, 9); // cycles 0-3
sub(7, 4, 3); // cycles 1-4
```



Inferred transaction trace:

```
add(7, 2, 9); // cycles 0-3
sub(7, 4, 3); // cycles 1-4
```





Case Studies

Goal: demonstrate that the same Paso spec can be used to both drive designs and infer transactions

AXI-Stream

Ready-valid interface for stream processing (e.g. for packets)

arm

Wishbone

Open-source interconnect for on-chip communication

Real-World Protocol Bugs

RQ: Can our reconstructor help identify protocol violation bugs in real-world implementations?

Benchmark suite of bugs in open-source FPGA designs (we focus on 5 protocol violation bugs for AXI-Stream)

Debugging in the Brave New World of Reconfigurable Hardware

Jiacheng Ma
University of Michigan

Gefei Zuo
University of Michigan

Kevin Loughlin
University of Michigan

Haoyang Zhang
University of Michigan

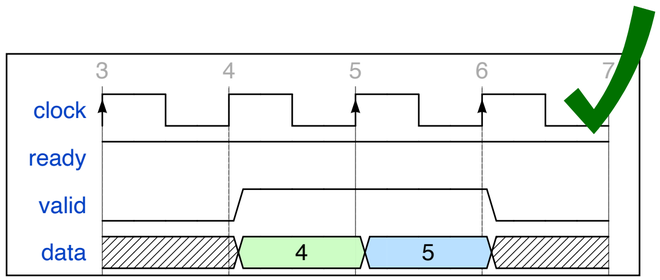
Andrew Quinn
University of California, Santa Cruz

Baris Kasikci
University of Michigan

ASPLOS'22

Real-World Protocol Bugs

```
prot send<DUT: AXIS>(…) { … }
prot recv<DUT: AXIS>(…) { … }
```

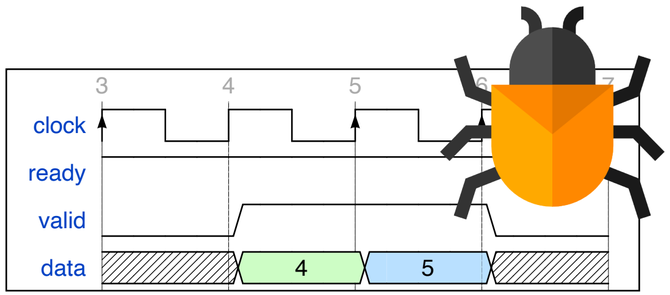


Reconstructor

```
Inferred trace:
  send_data(4); // cycle 4-5
  send_data(5); // cycle 5-6
  …
```

Real-World Protocol Bugs

```
prot send<DUT: AXIS>(…) { … }
prot recv<DUT: AXIS>(…) { … }
```



Reconstructor

```
Error: transaction failed
at 100ns!
…
```

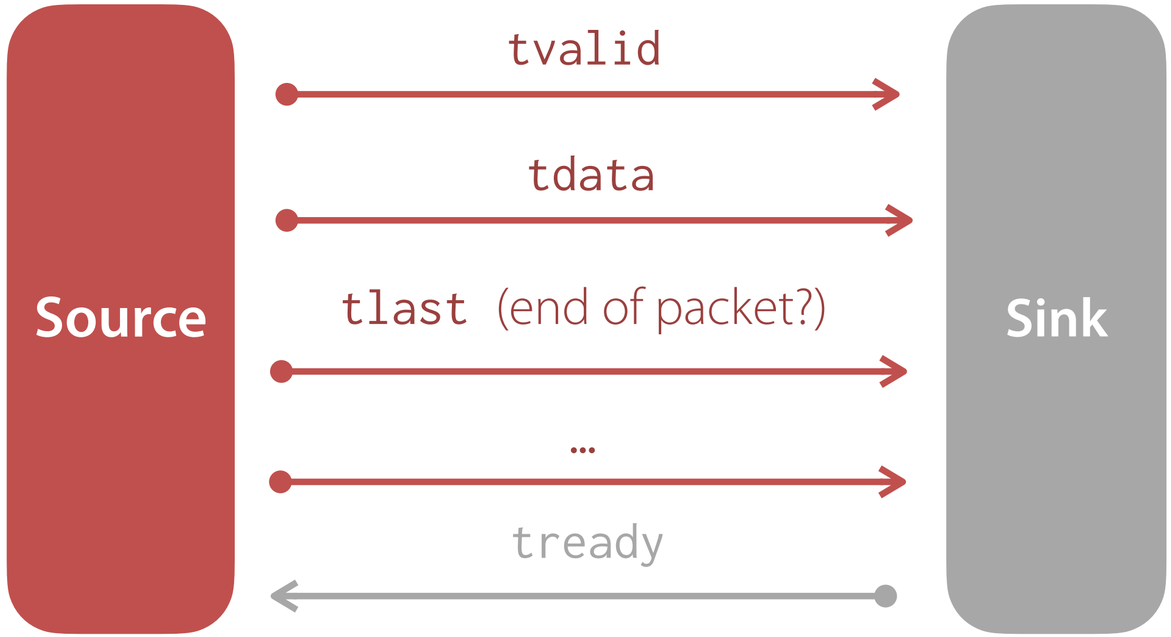
Example Protocol Violation Bug (1)

(`axi-stream-s2` from Ma et al.'s ASPLOS '22 artifact)

Debugging in the Brave New World of Reconfigurable Hardware

Jiacheng Ma, University of Michigan
Gefei Zuo, University of Michigan
Kevin Loughlin, University of Michigan
Haoyang Zhang, University of Michigan
Andrew Quinn, University of California, Santa Cruz
Baris Kasikci, University of Michigan

Buggy DUT:
Xilinx AXI-Stream source
(transmits packets)



AXI-Stream Spec (§2.2.1):
when **`tvalid`** is high, **`tlast`** must remain stable (i.e., unchanged) until a ready-valid handshake occurs

Example Protocol Violation Bug (1)

(`axi-stream-s2` from Ma et al.'s ASPLOS '22 artifact)

Xilinx AXI-Stream manager fails to hold `tlast` stable when `tvalid` is high

(end of packet signal)

Reconstructor error on buggy waveform

```
error: [recv] transaction failed at 1037.5 ns!
  assert_eq(DUT.tlast, is_last);
  ^^^^^^^^^^
expected DUT.tlast = 0 but got DUT.tlast = 1
```

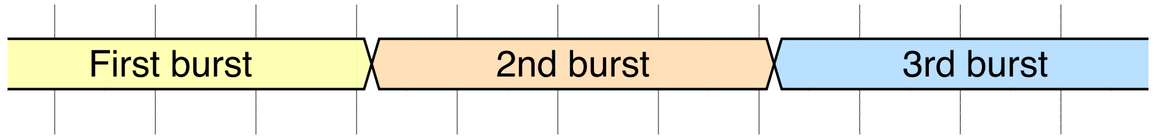
Transactions inferred from fixed waveform

```
reset();                        // time: 0ns -> 12.5ns
recv(data=1, is_last=0); // time: 12.5ns -> 887.5ns
…
recv(data=6, is_last=0); // time: 987.5ns -> 1012.5ns
```

Example Protocol Violation Bug (2)

Wishbone supports burst mode with “wrapped” address increments

(i.e., automatically increment addresses during each transfer, but ensure they’re aligned)



DUT: Wishbone SRAM designed by Antmicro

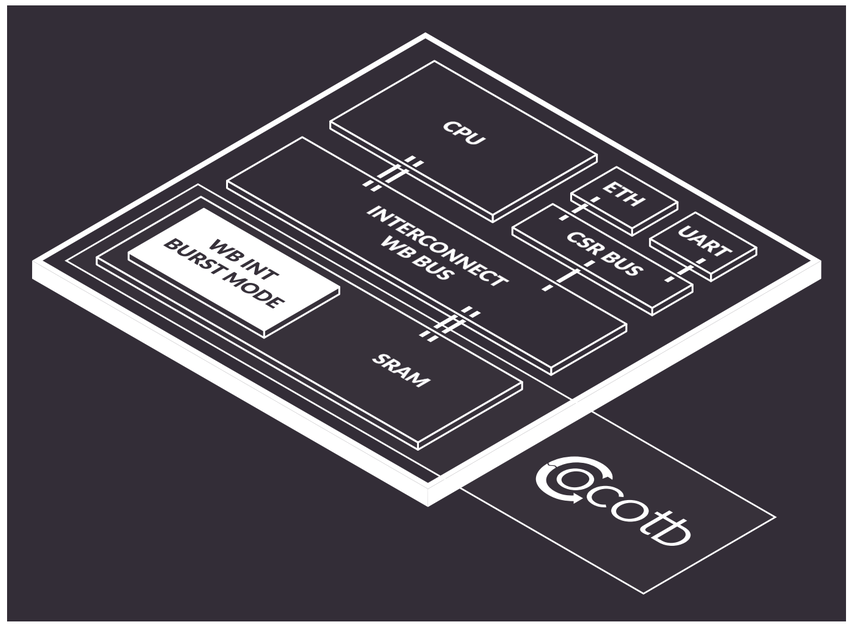
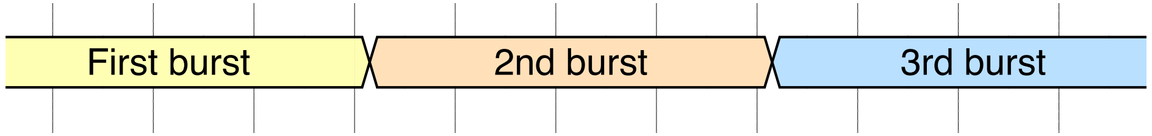


Figure from Antmicro

https://antmicro.com/blog/2022/05/faster-soc-interconnects-with-test-driven-fpga-development-and-cocotb

Example Protocol Violation Bug (2)

Antmicro Wishbone SRAM address computation bug in WB burst mode



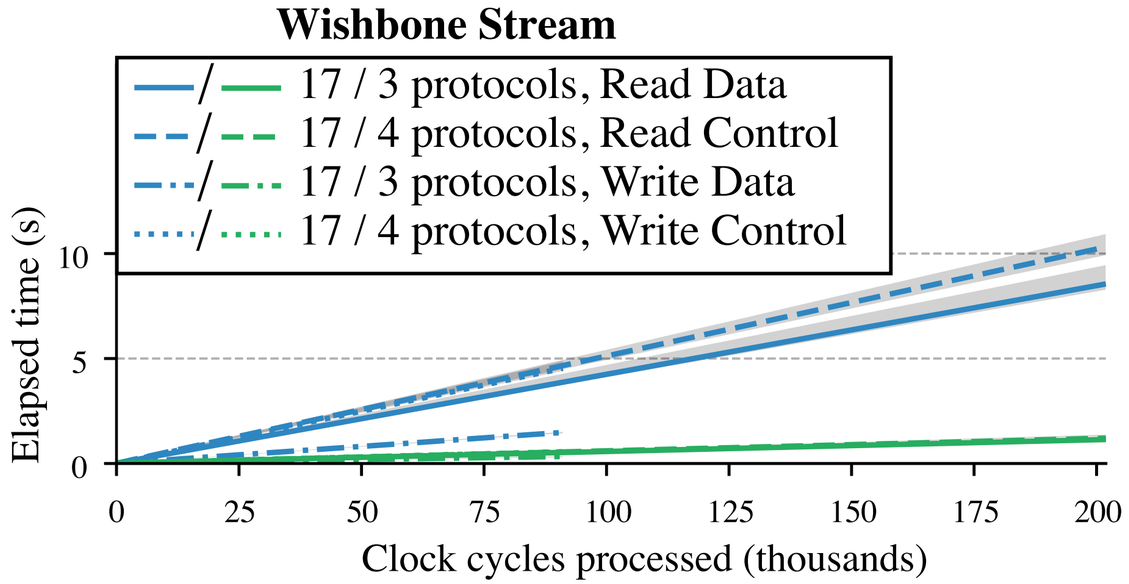
Reconstructor Error

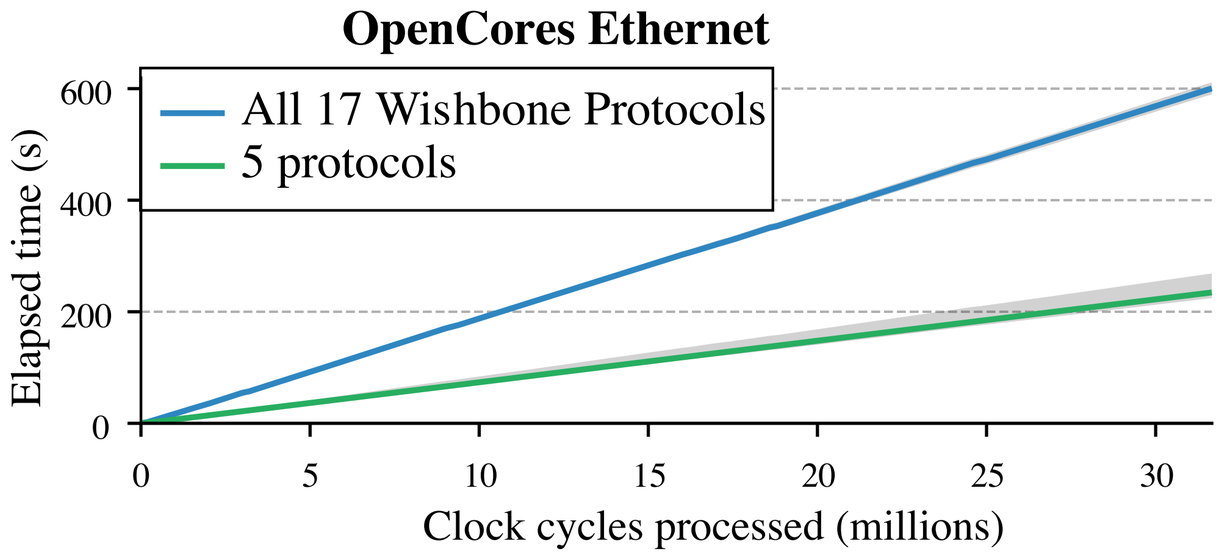
```
error: [read_burst_increment_wrap] transaction failed at 500ns!
  DUT.addr := start_addr[31:5] ## (start_addr[4:2] + iter_count()) ## start_addr[1:0];
  ^^^^^^^^
expected DUT.addr = 0x04000000 but got 0x04000008
```

Reconstructor Performance

Reconstructor scales to long waveforms (>75k cycles)
collected from open-source Wishbone projects

Performance scales proportionally to no. of clock cycles
in waveform + no. of protocol specs

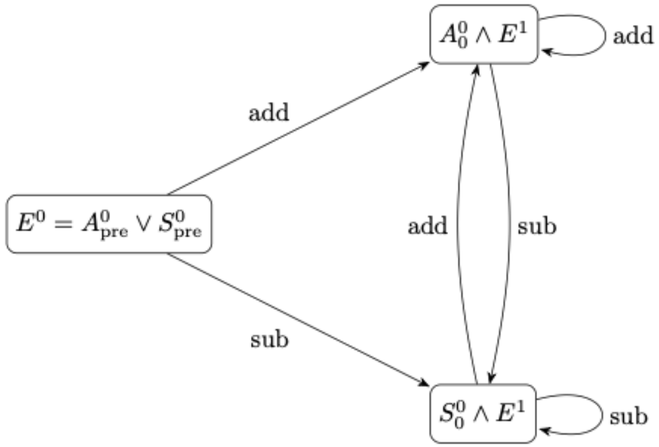




Future Work

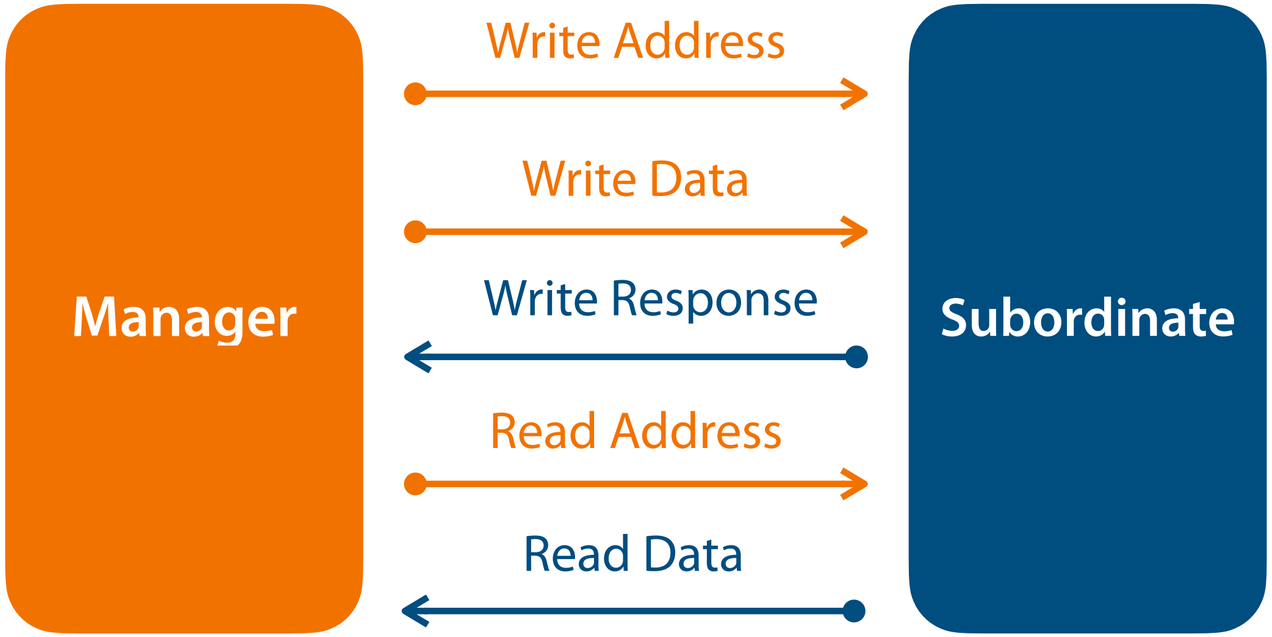
In-FPGA Runtime Monitoring

Compile protocols to automata, which can be represented as FSMs in RTL (ongoing work by Nikil)



Support full AXI / AXI-Lite

Involves reasoning about dependencies between transactions on different channels

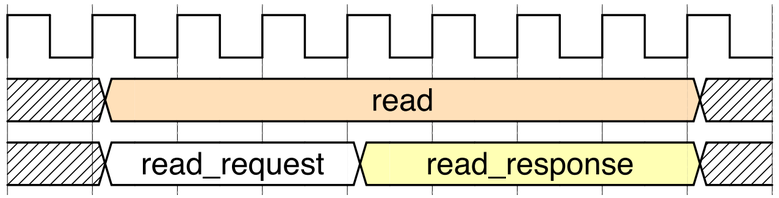


Future work: *Transaction-level* visualizations

1. Augmented waveform viewer

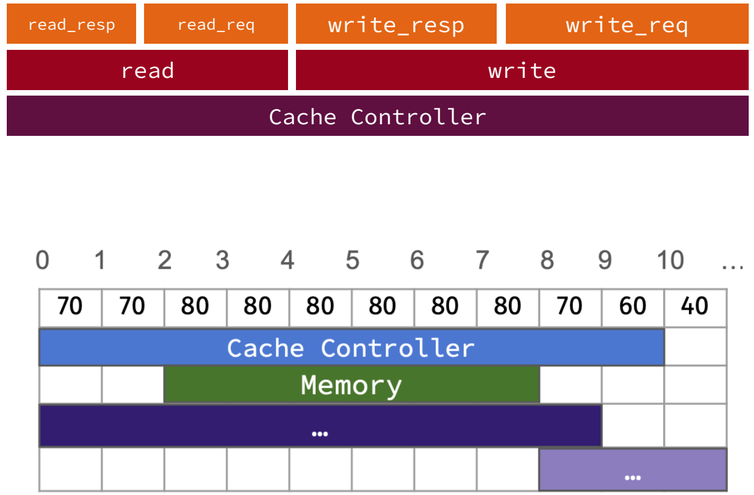
Visualize inferred transactions in the Surfer waveform viewer

[Skarman et al. CAV '25]



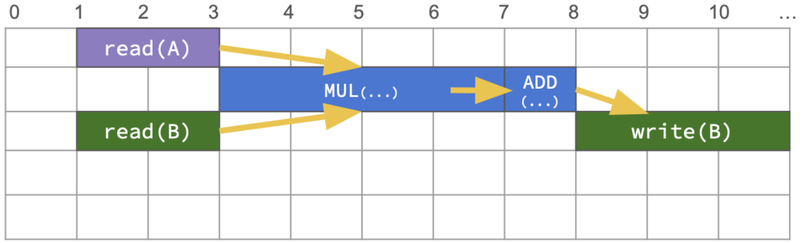
2. Transaction profiler

Visualizes performance bottlenecks



3. Dependency tracker

Visualizes & monitors inter-transaction dependencies



Provides engineers with high-level transaction info while integrating with their existing workflows

Applicable at any level of hardware description (RTL, HLS)

Assists engineers with dependency tracking at a higher abstraction level

Thanks!

ernest
@cs.cornell.edu

cucapra/protocols

Conclusion: Hardware communication protocols can be specified as programs!
Using our DSL, the same protocol spec can be used to both run tests & infer transactions.

Ernest
Ng

Nikil
Shyamsunder

Francis
Pham

Adrian
Sampson

Kevin
Laeufer