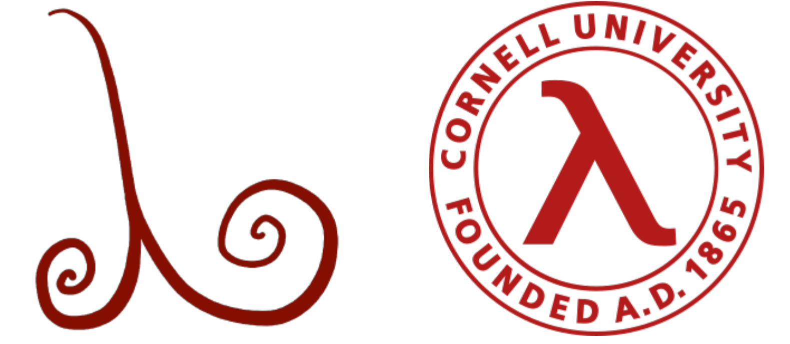


Capra lab @ Cornell

Specifying Hardware Communication as Programs

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

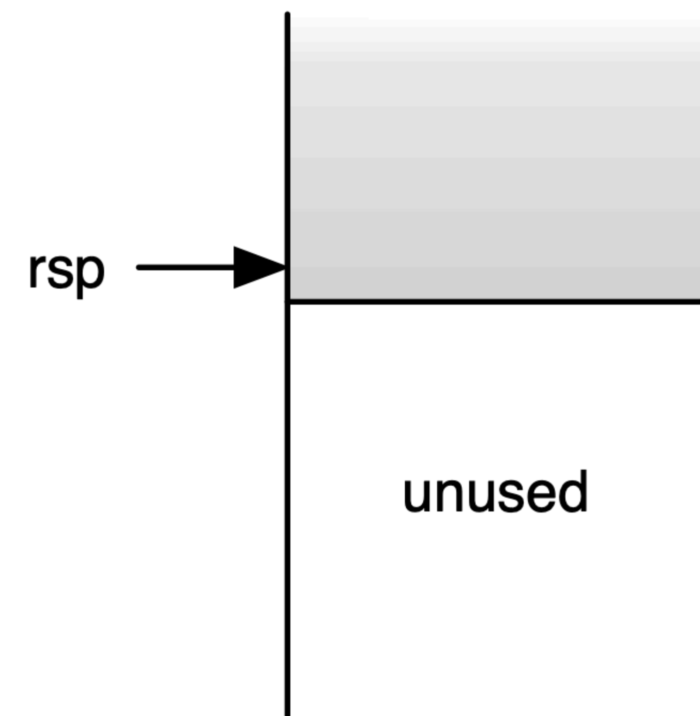
MIT FLAME Lab Seminar
July 8, 2026

Calling Conventions in Software

Standardized contract about how to invoke functions

Calling Conventions in Software

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



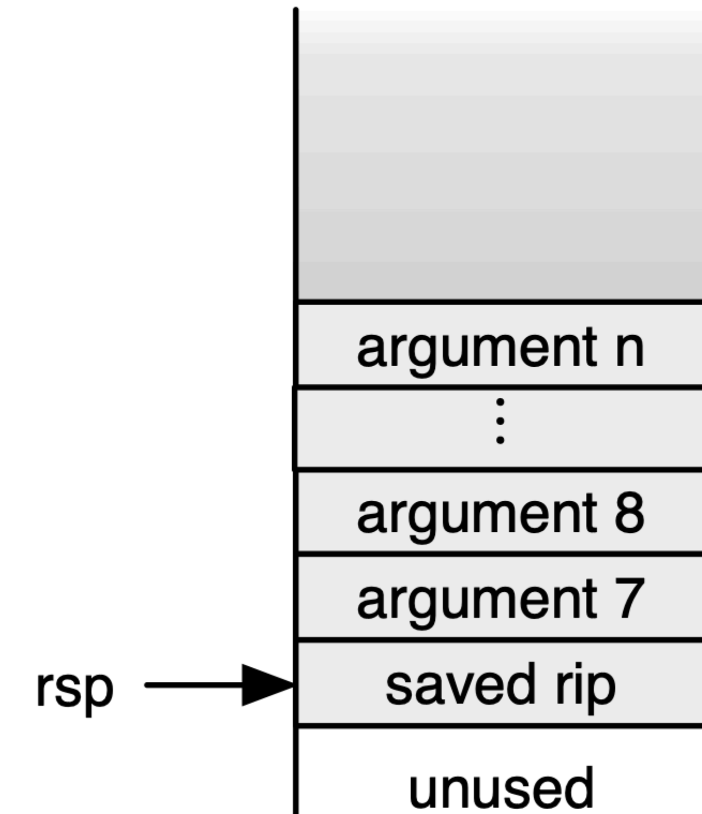


Figure from Andrew Myers

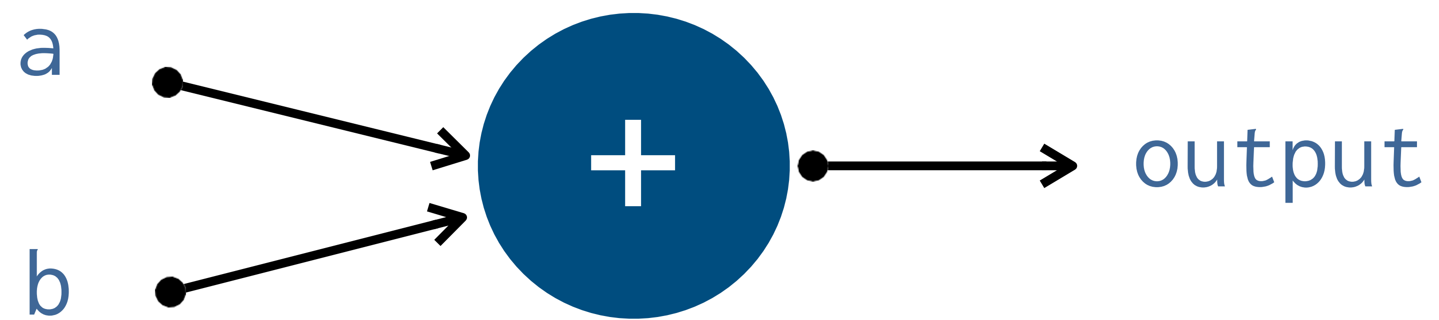
Calling Conventions in *Hardware*?

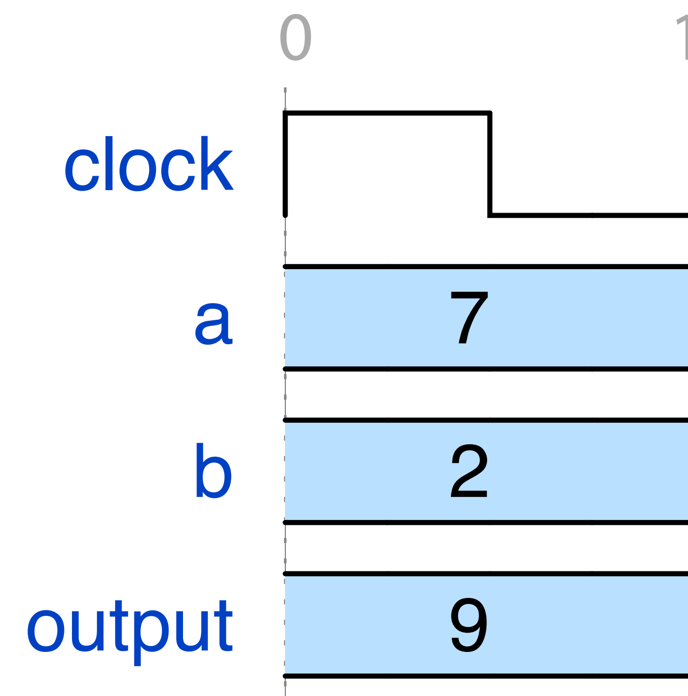
Calling Conventions in *Hardware*?

There is no one way to “call” a HW module!

“Calling” an adder in hardware

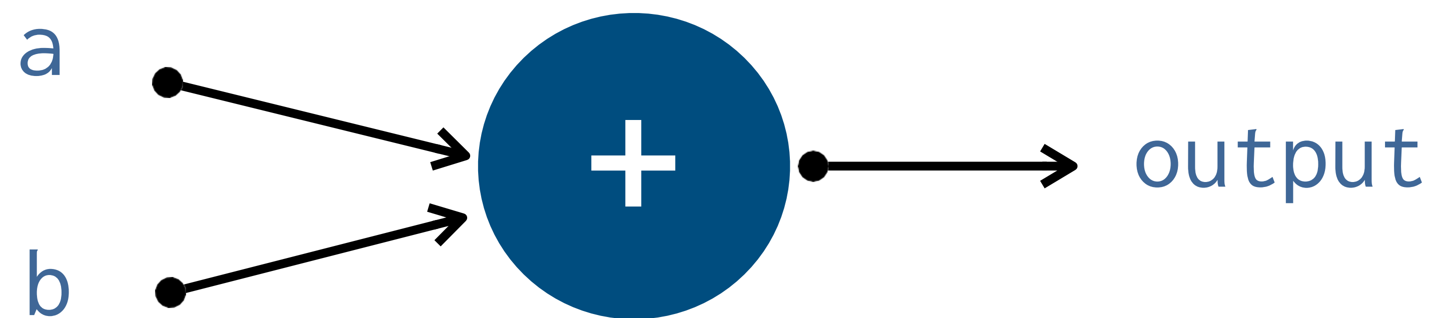
Combinational Adder

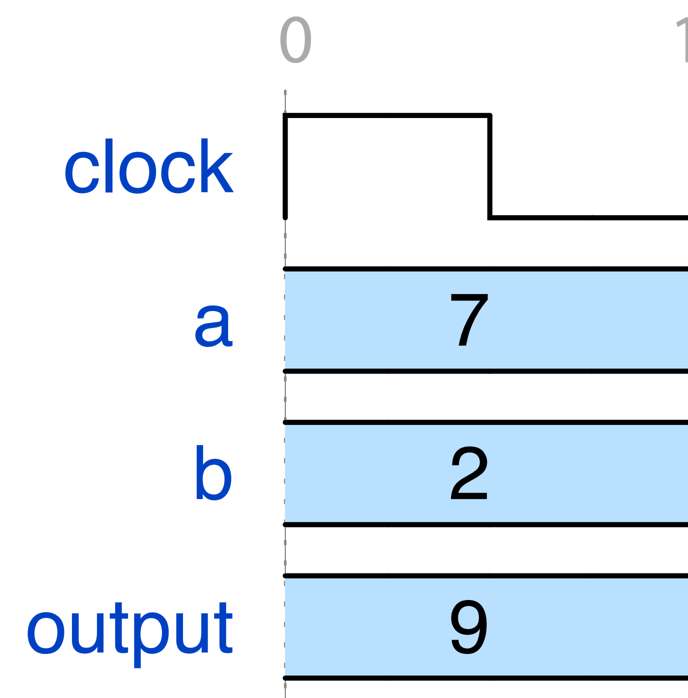




“Calling” an adder in hardware

Combinational Adder



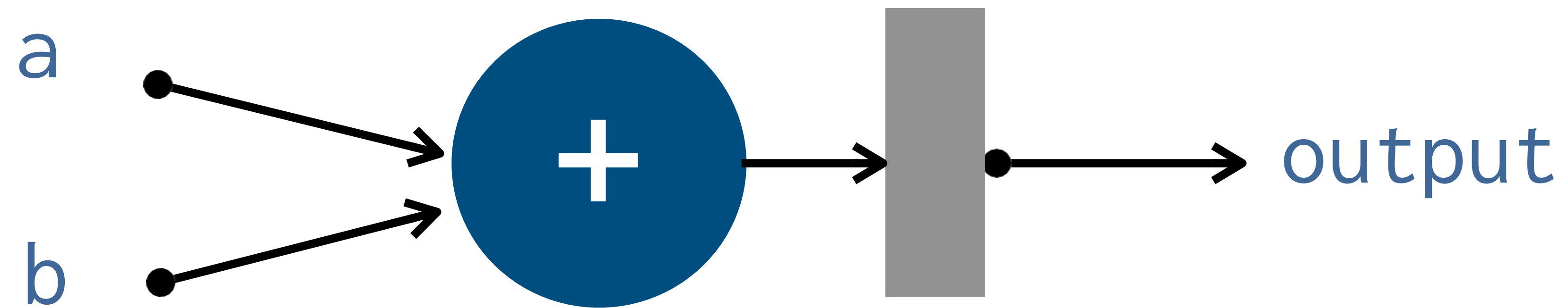


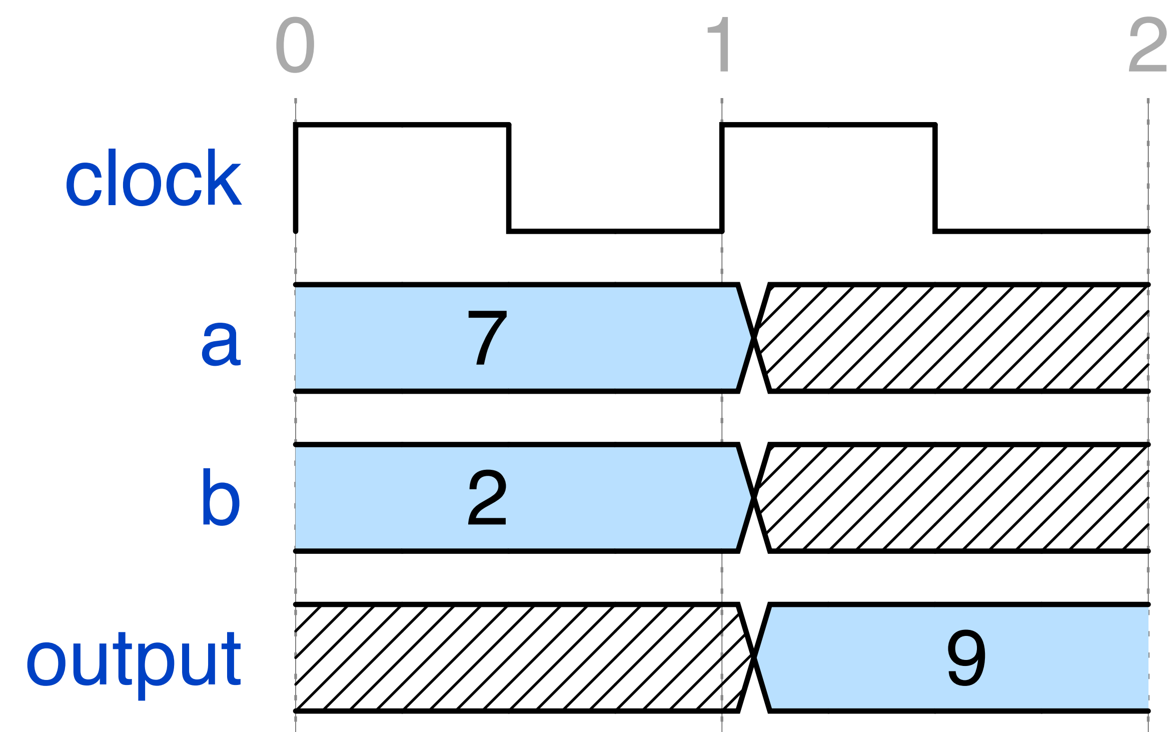
“If I drive `a = 7`, `b = 2`,

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

“Calling” an adder in hardware

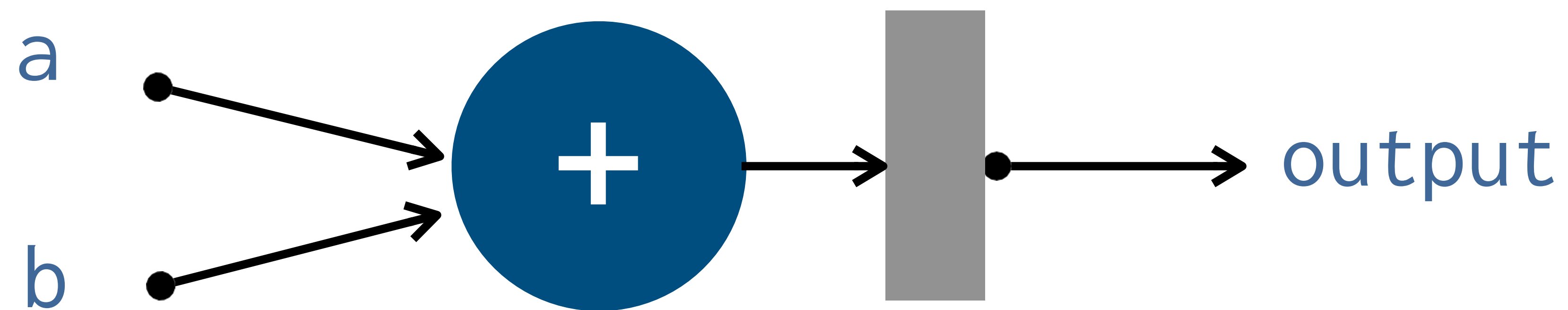
Sequential Adder

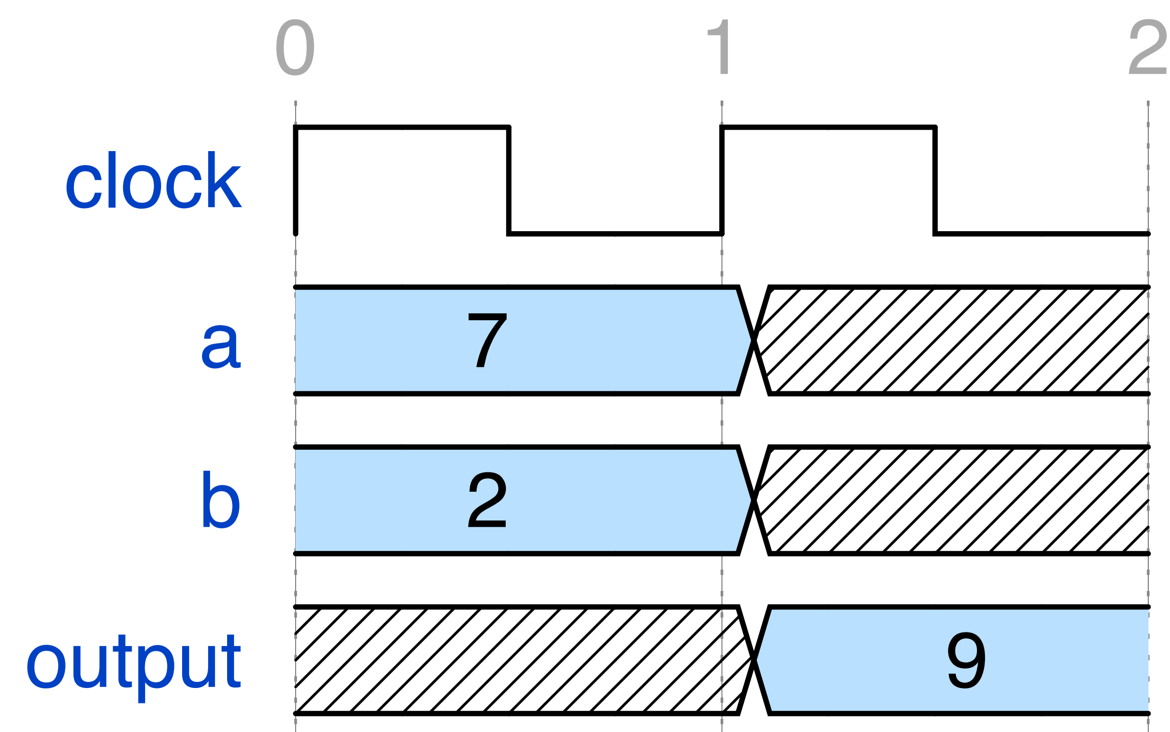




“Calling” an adder in hardware

Sequential Adder





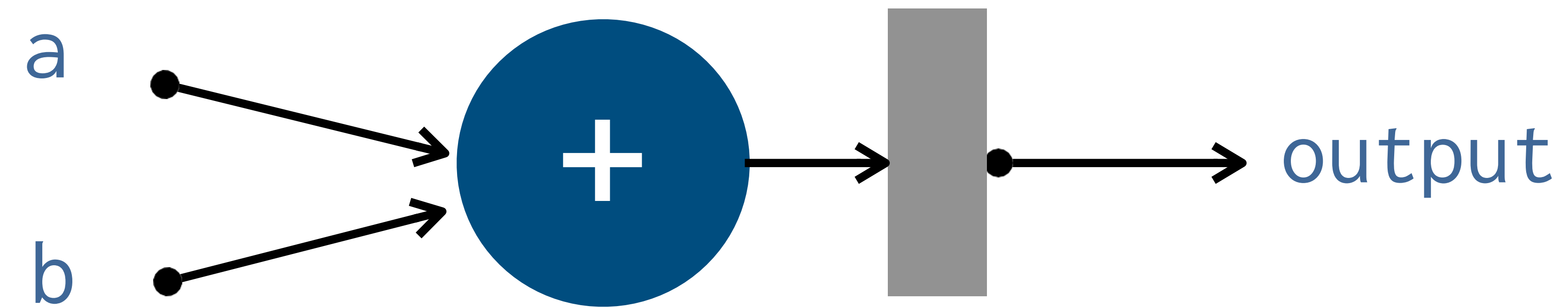
“If I drive `a = 7`, `b = 2`,

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

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

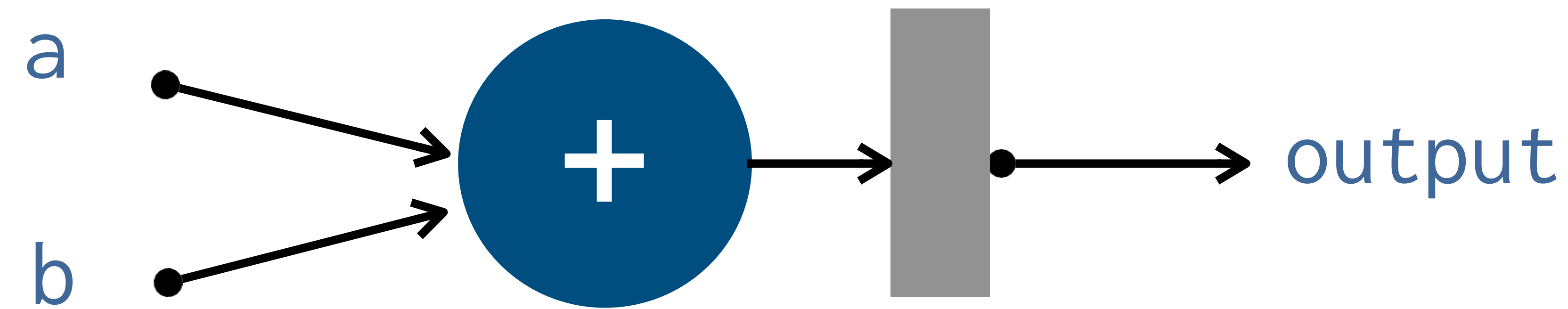
The abstraction gap in hardware

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



The abstraction gap in hardware

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

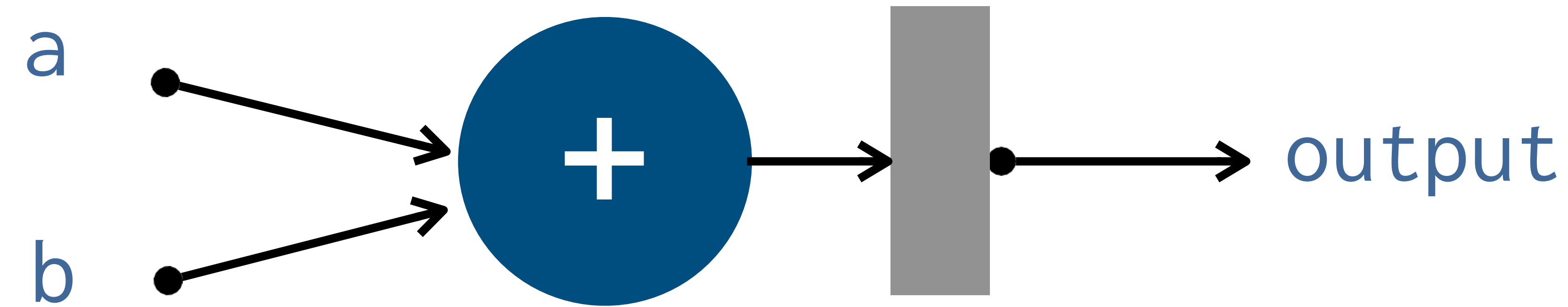


Transactions

"Adding 7 and 2
results in 9"

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

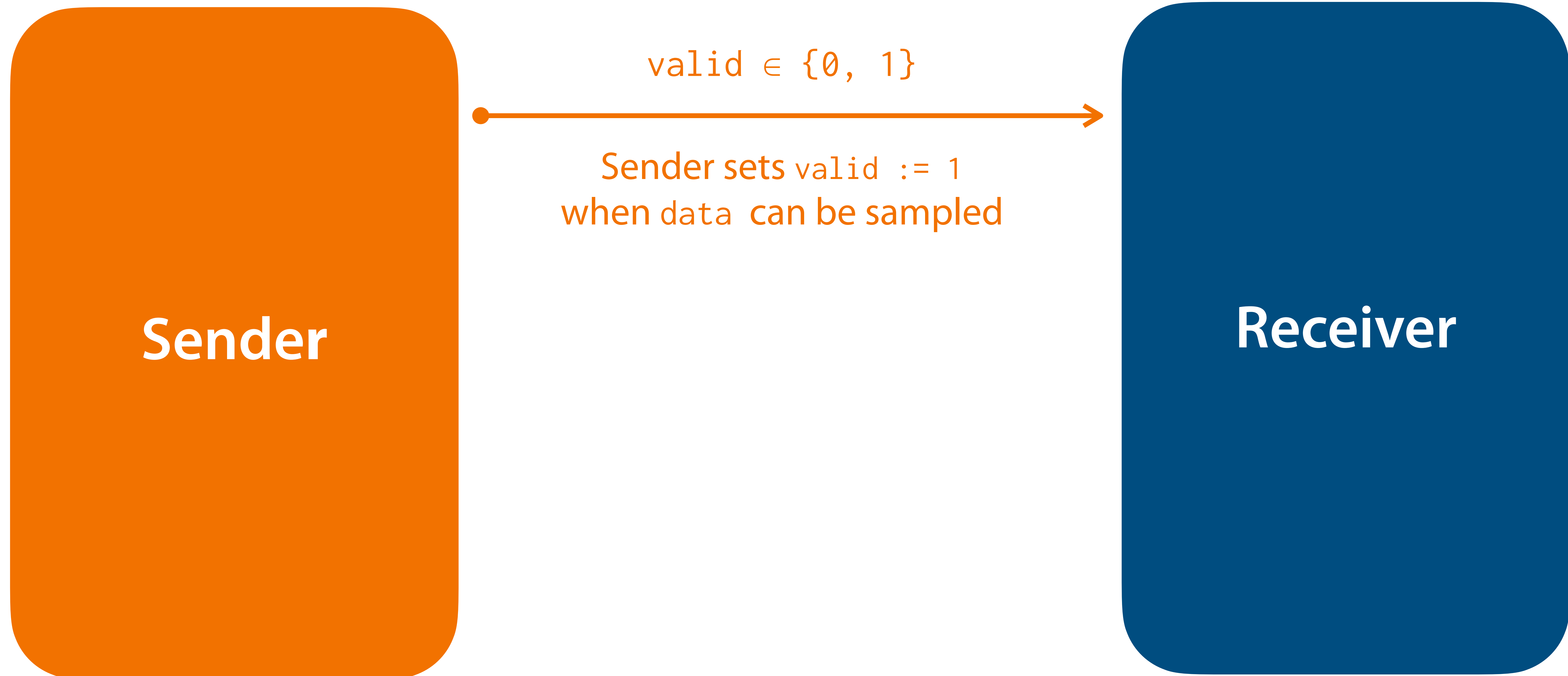
Ready-Valid Handshake

Mediates data transfer between a sender & a receiver

Receiver

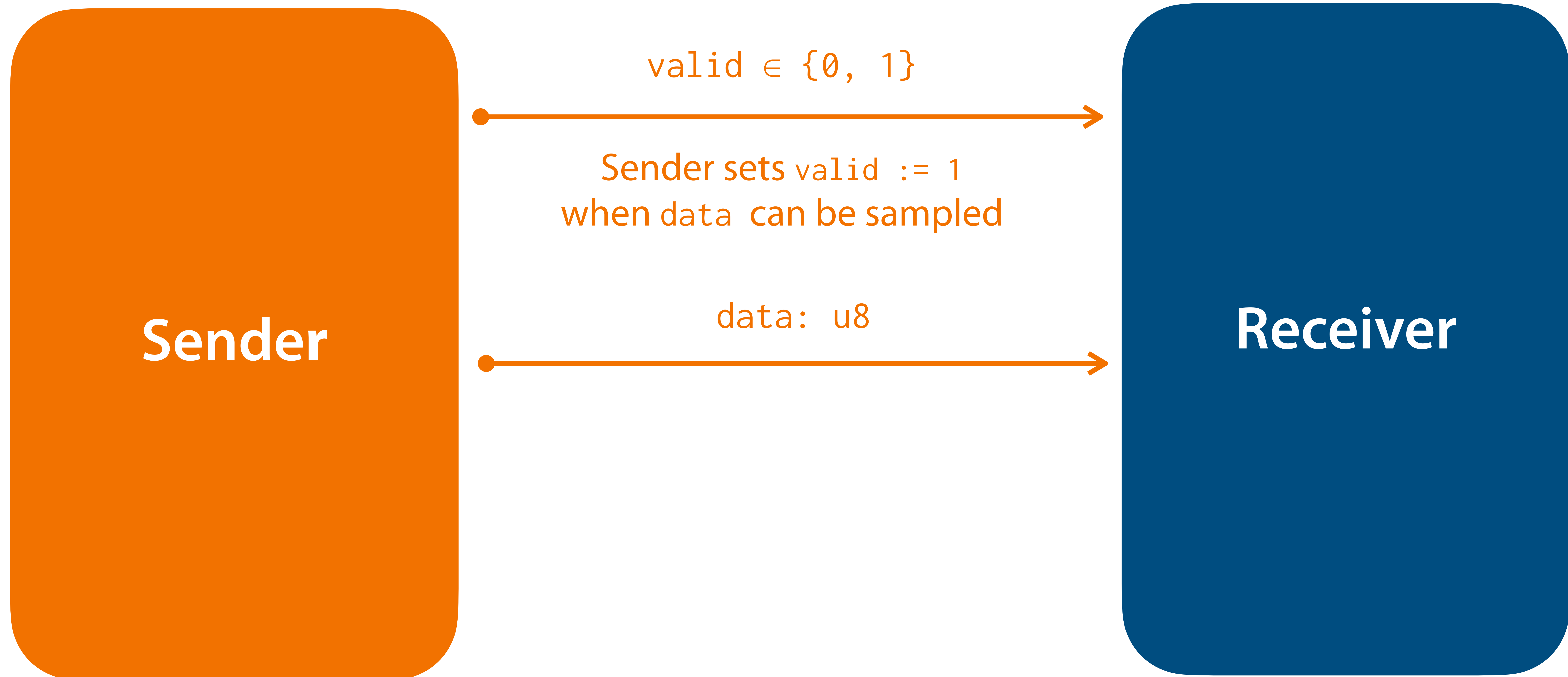
Ready-Valid Handshake

Mediates data transfer between a sender & a receiver



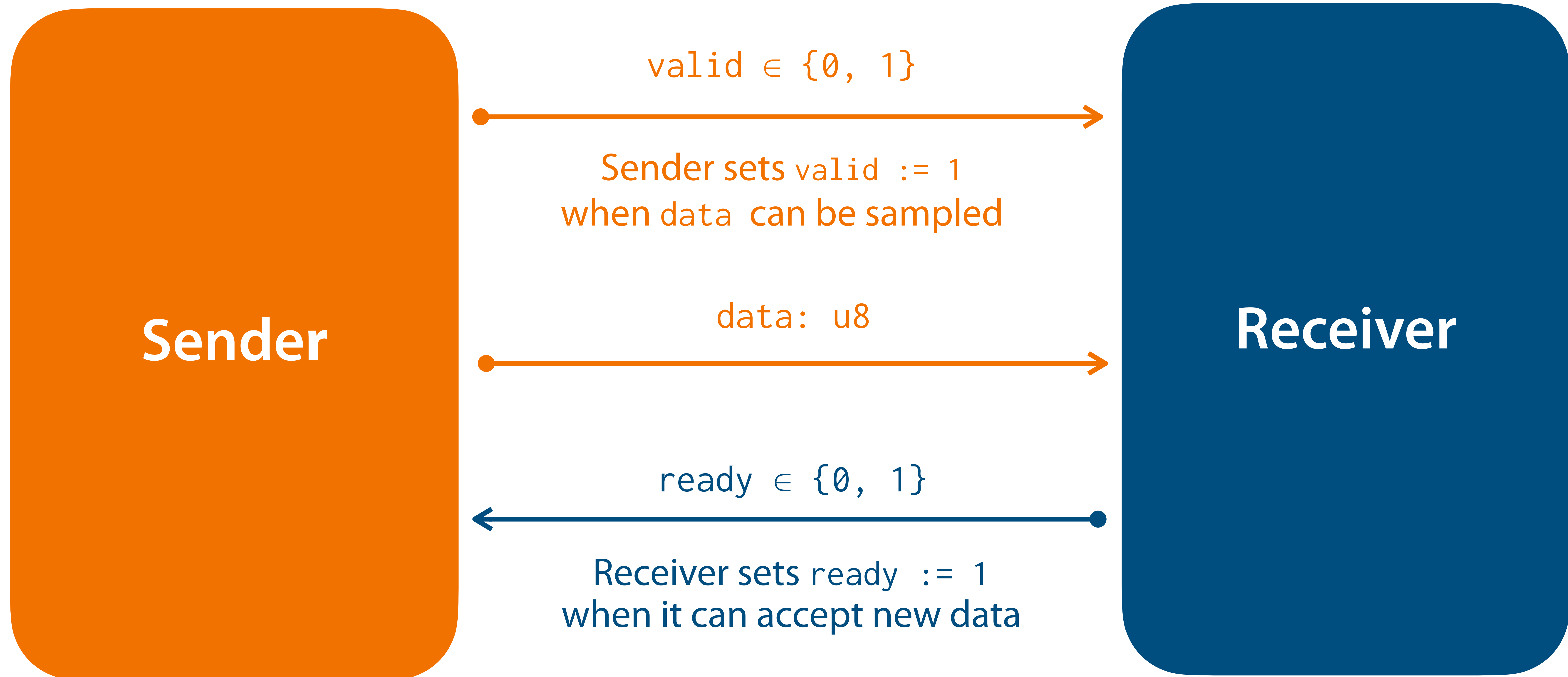
Ready-Valid Handshake

Mediates data transfer between a sender & a receiver



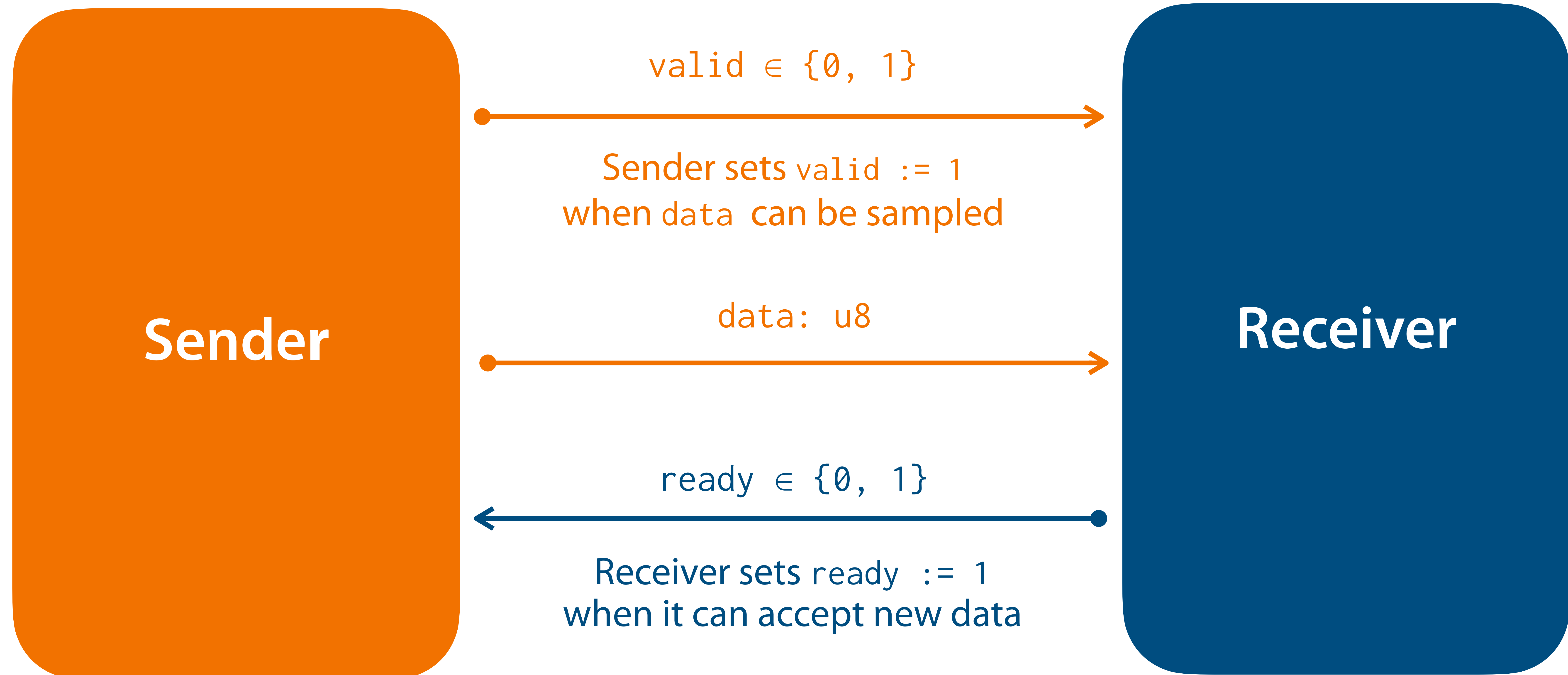
Ready-Valid Handshake

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:

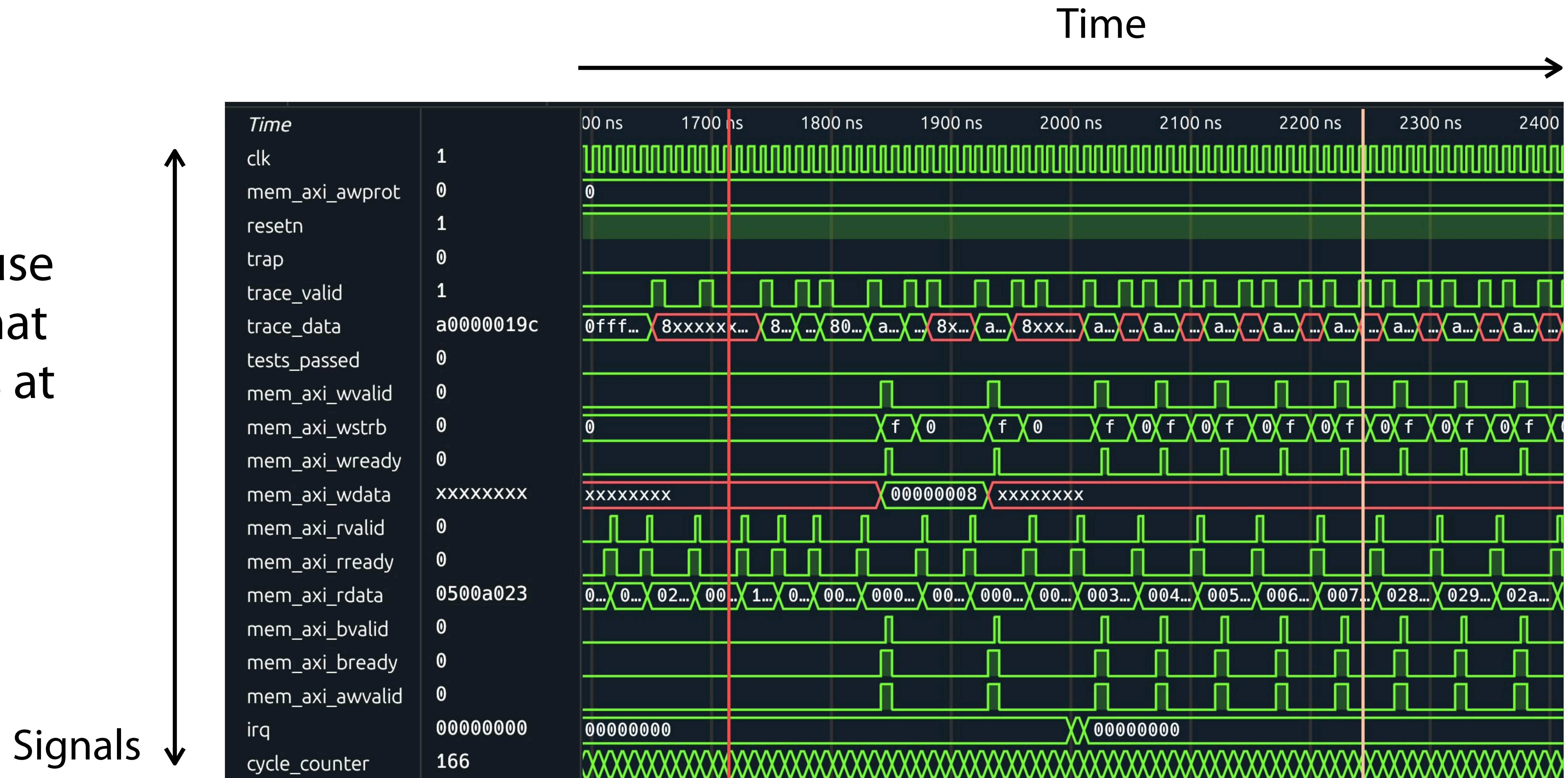
These protocols are often stated implicitly (in English), making testing & debugging hard!

Problem:

These protocols are often stated implicitly (in English), making testing & debugging hard!

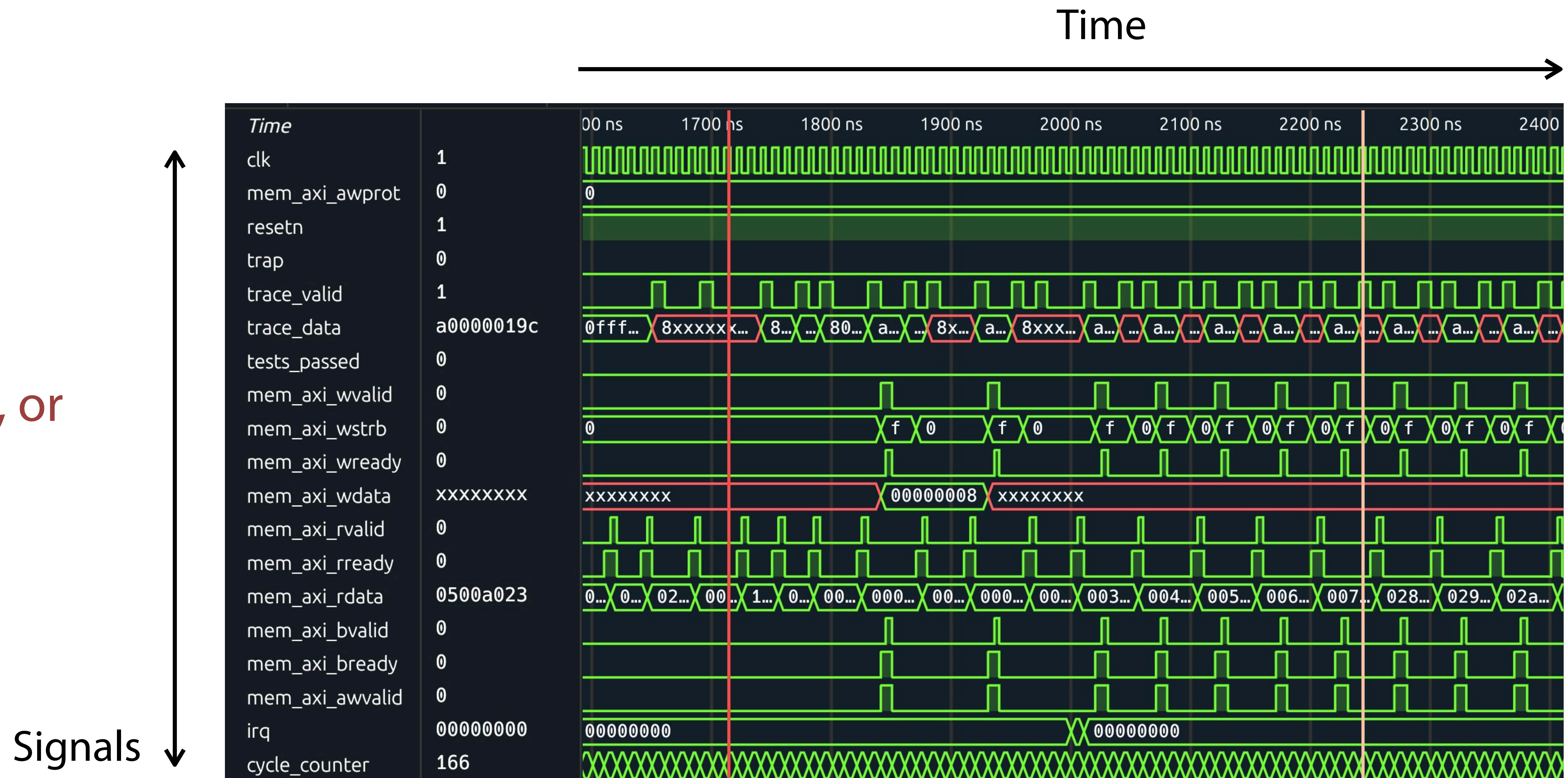
Debugging hardware is hard

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



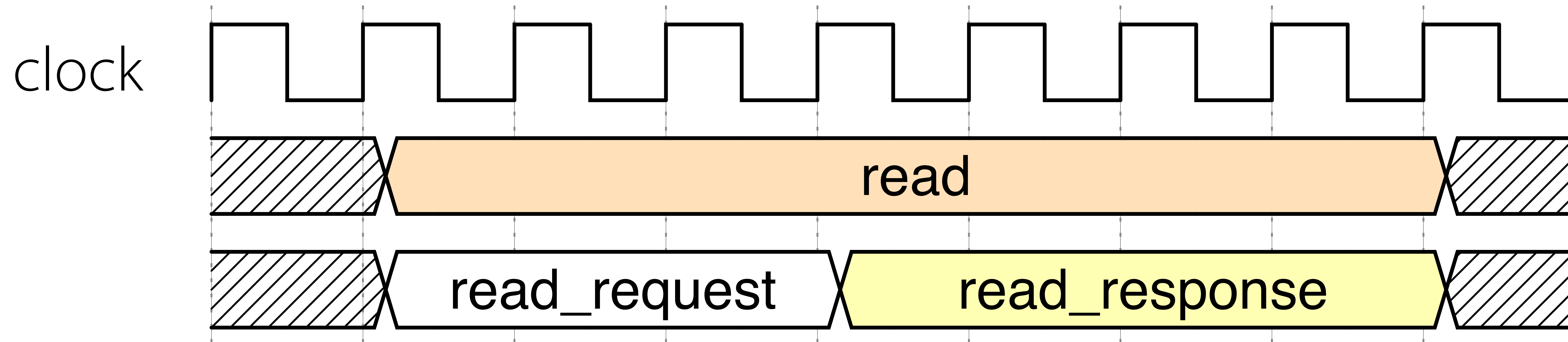
Debugging hardware is hard

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

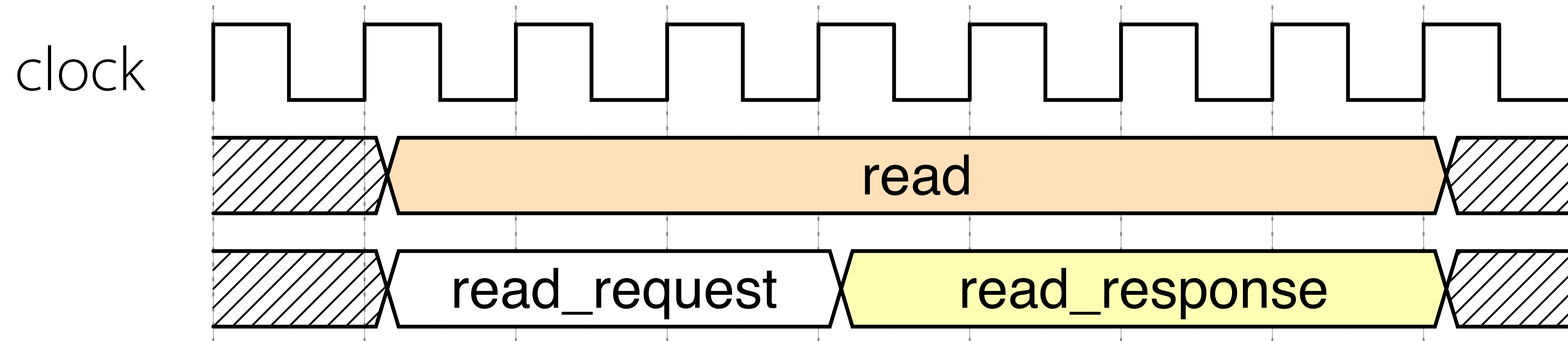


What if we had waveform viewers that looked like this instead?

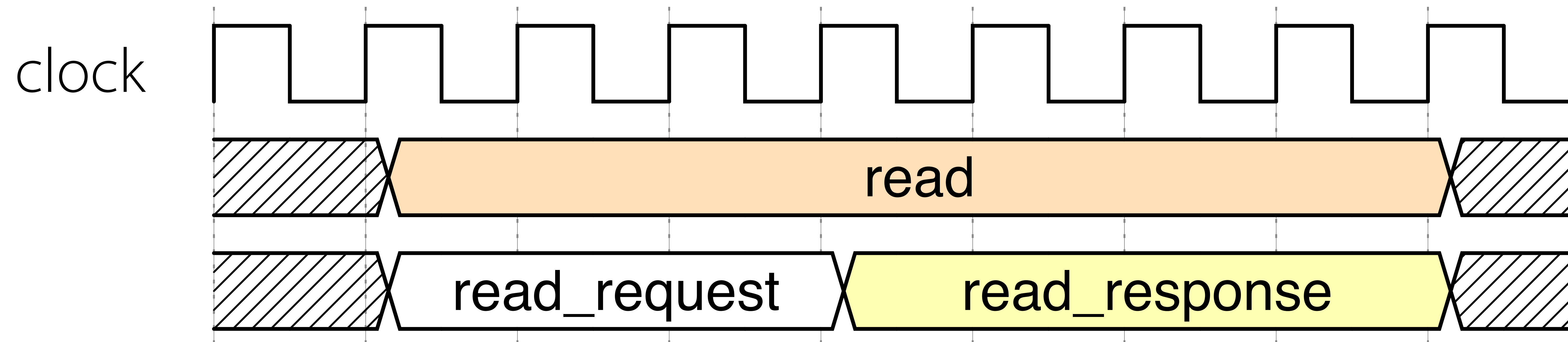
What if we had waveform viewers that looked like this instead?



Problem: need to ***reconstruct*** transactions from signals



Problem: need to ***reconstruct*** transactions from signals



I'll show you later how to do this!

Thesis

Thesis

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

Thesis

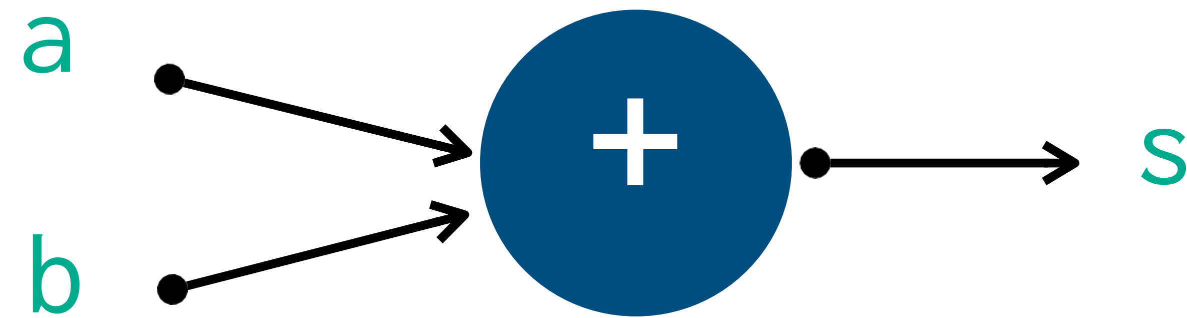
1. Hardware communication protocols can be specified as **programs**! We design a DSL, Paso, which enables this.

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

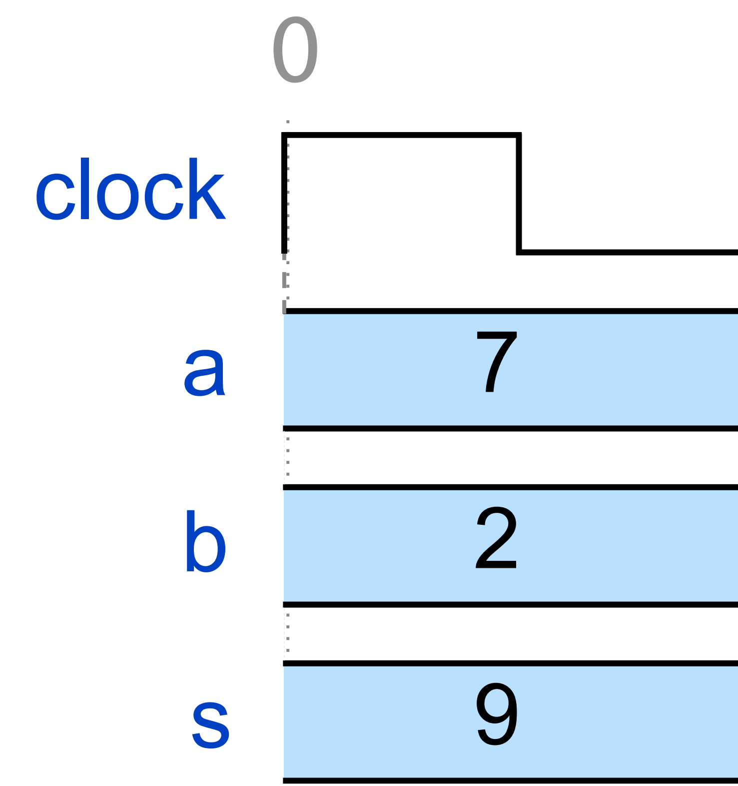
Paso by example:
Two protocol specs for an adder

A combinational adder

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



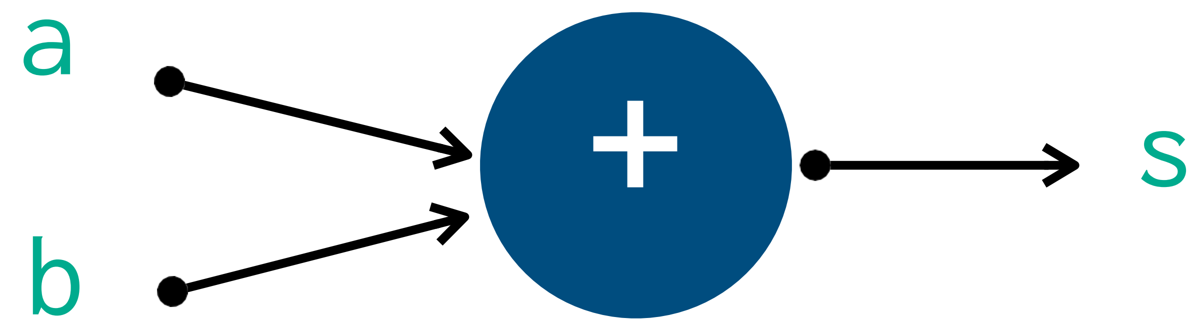
add(7, 2, 9)



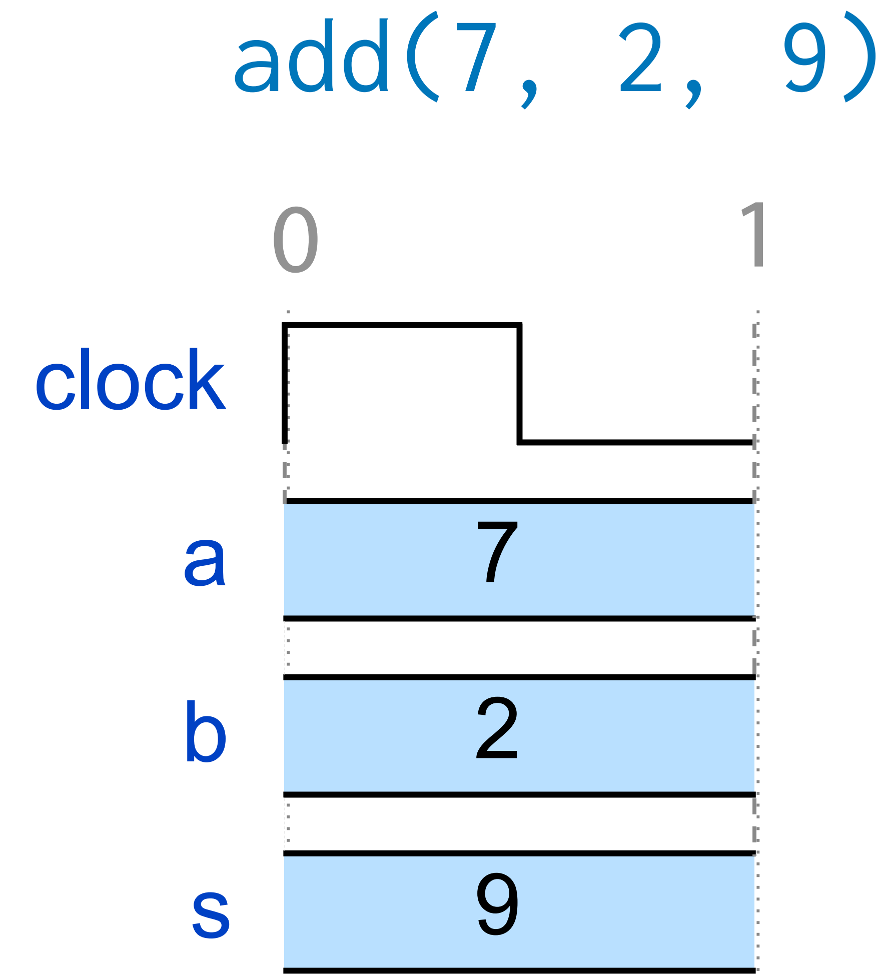

```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

A combinational adder

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



`add(7, 2, 9)`

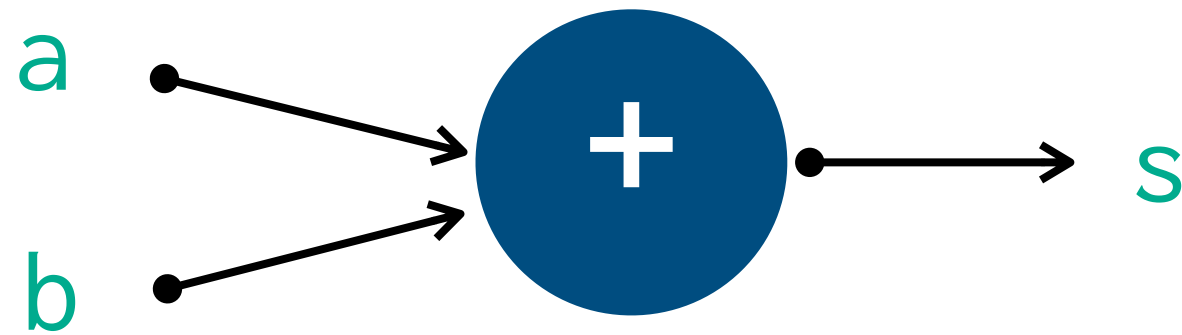


Notation: We specify expected inputs & outputs to the transaction, akin to a unit test

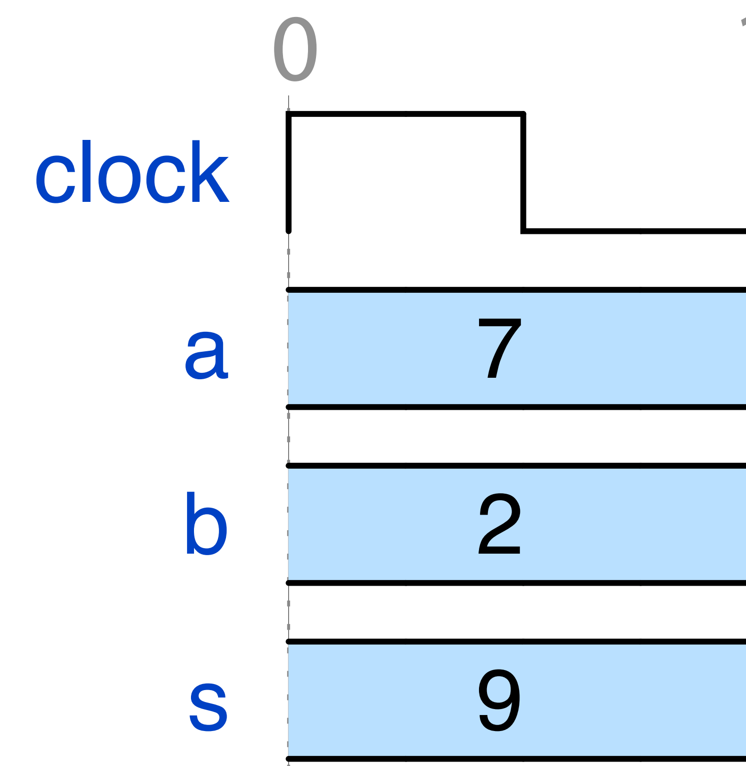
```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

A combinational adder

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



`add(7, 2, 9)`

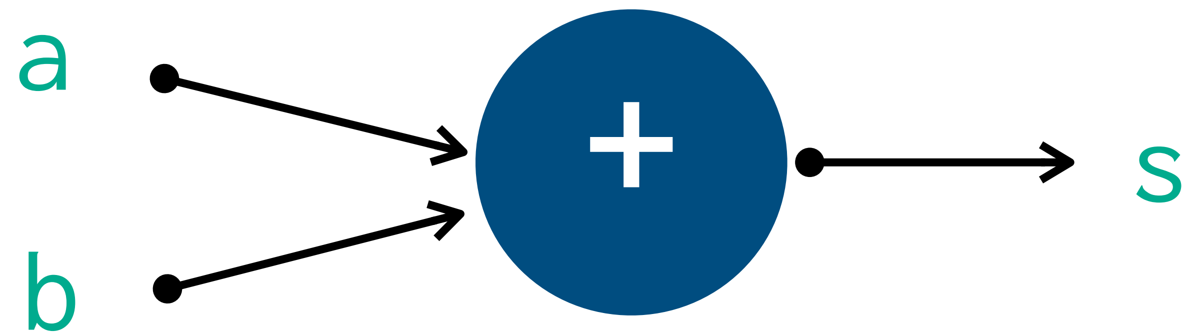


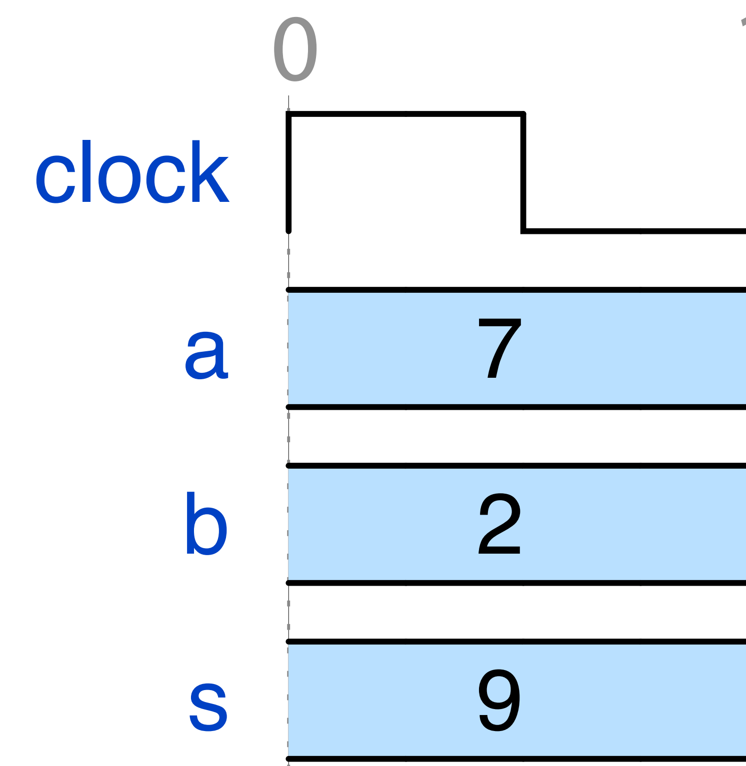

```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

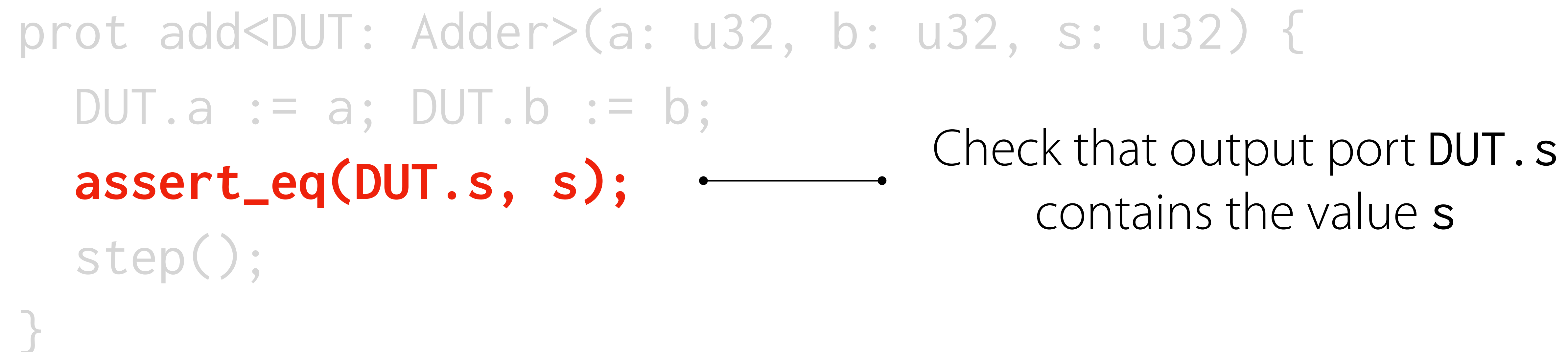
Apply values onto input ports `DUT.a` & `DUT.b`

A combinational adder

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



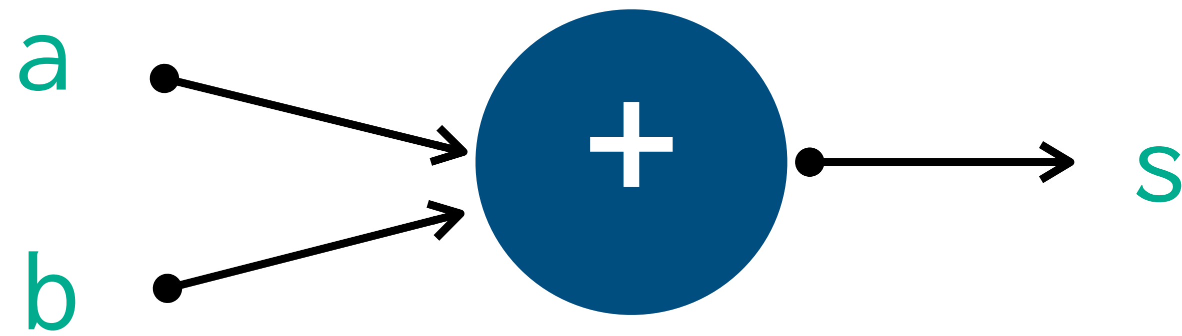




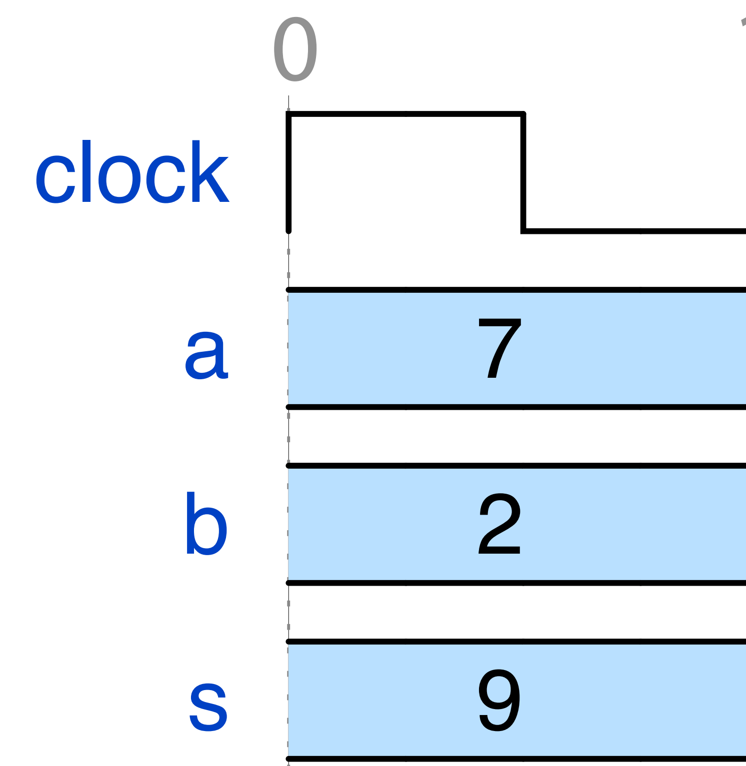
Check that output port `DUT.s` contains the value `s`

A combinational adder

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



add(7, 2, 9)

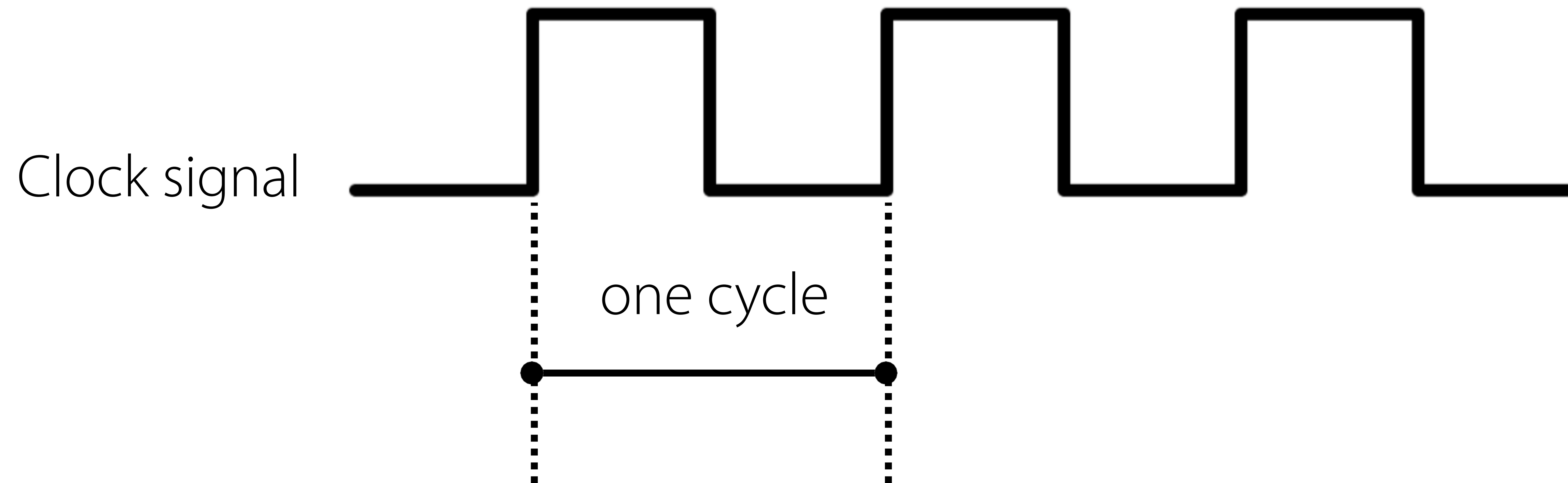



```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  assert_eq(DUT.s, s);
  step();
}
```

step(); — Finish clock cycle

Aside: the `step()` primitive

`step()` advances the clock by one cycle



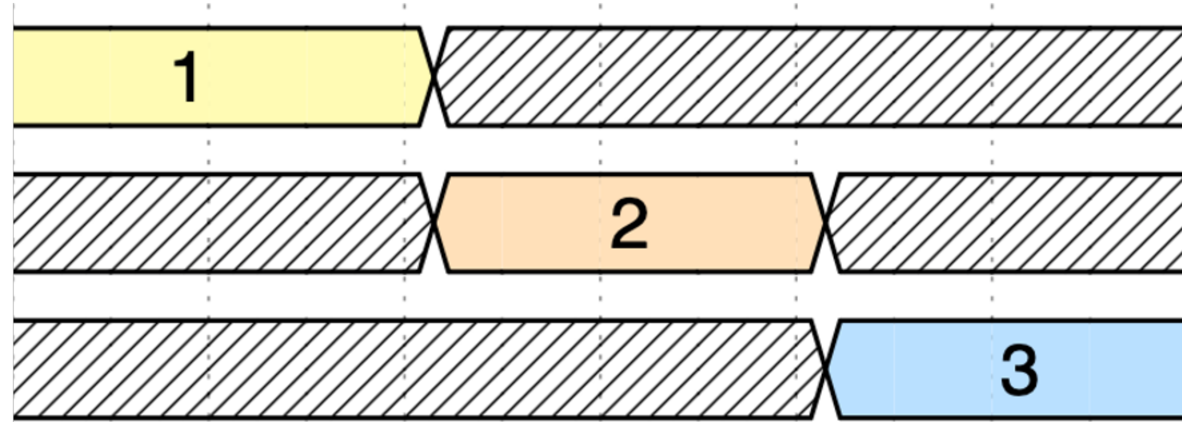
Can we optimize our adder?

Yes, through **pipelining**!

Yes, through **pipelining**!

Sequential

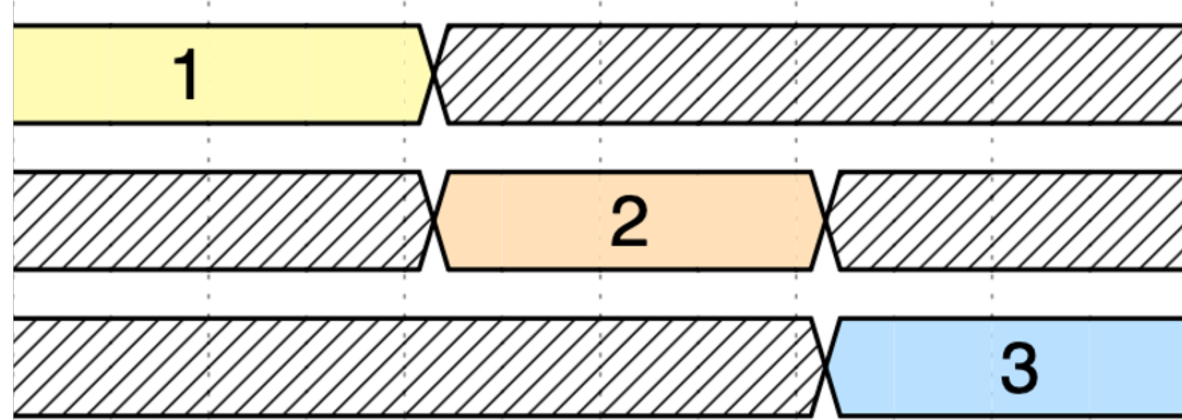
Processes **one** input at a time



Yes, through **pipelining**!

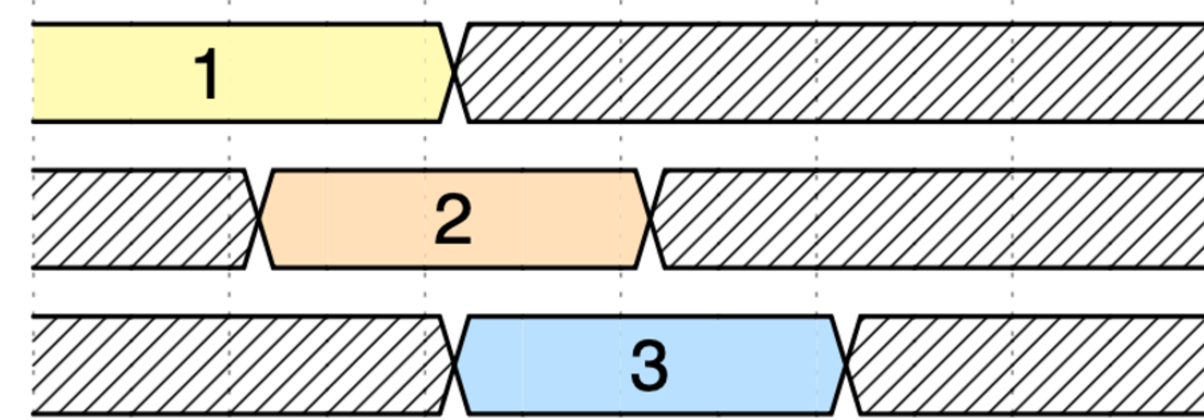
Sequential

Processes **one** input at a time



Pipelined

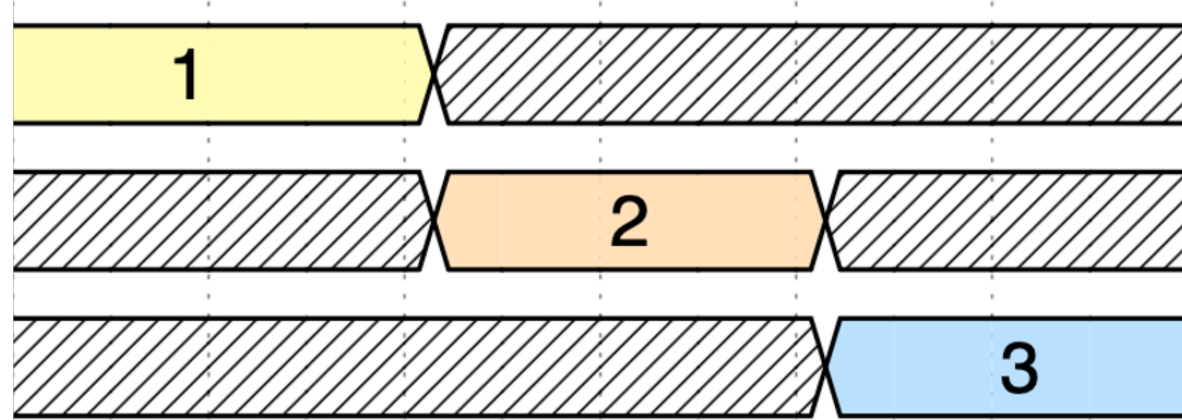
Processes **multiple** inputs at a time



Yes, through **pipelining**!

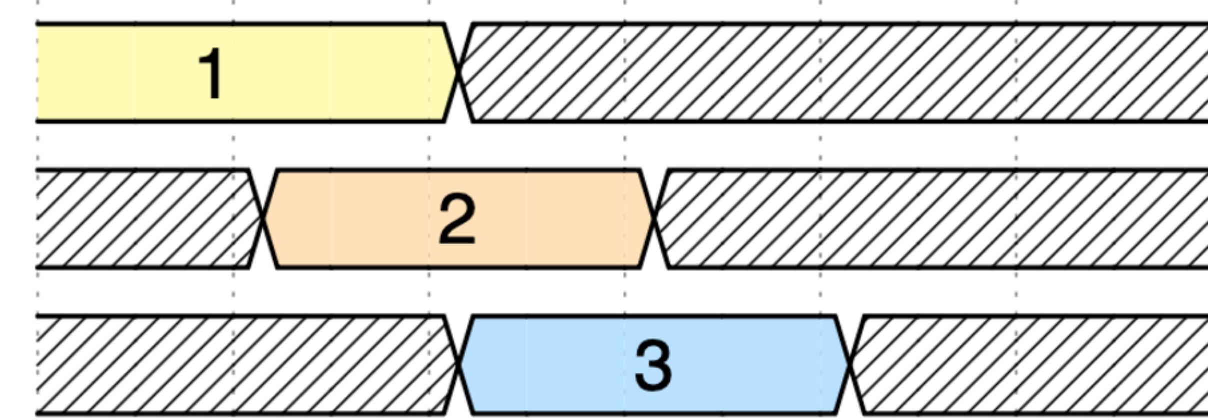
Sequential

Processes **one** input at a time



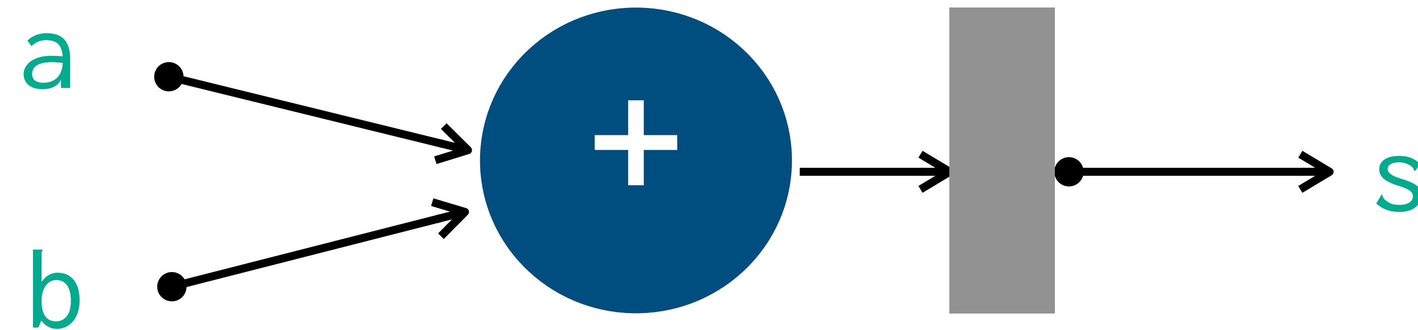
Pipelined

Processes **multiple** inputs at a time



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

A naïve attempt at a pipelined adder



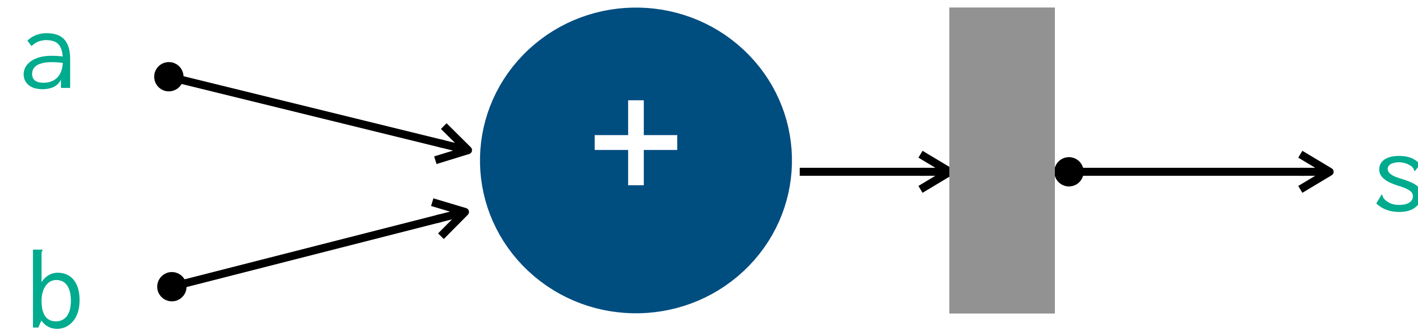
Combinational

```
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



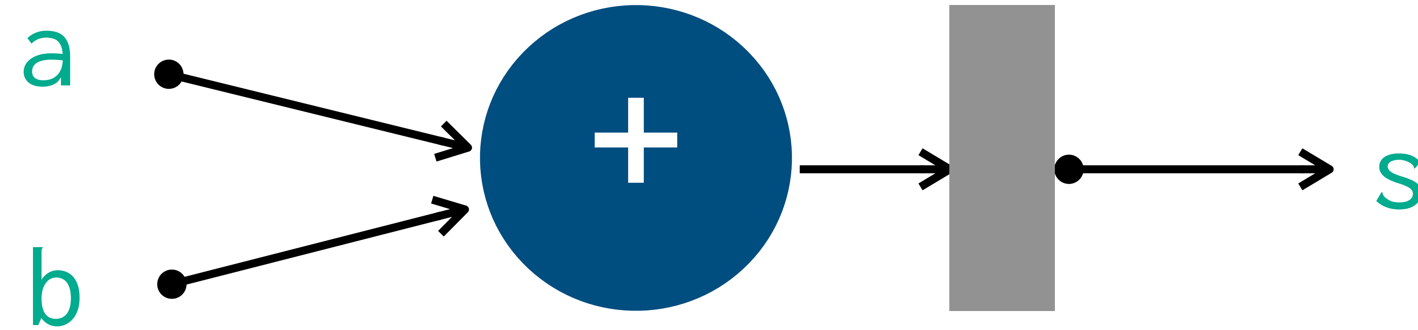
Combinational

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



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

Pipelined (naïve)

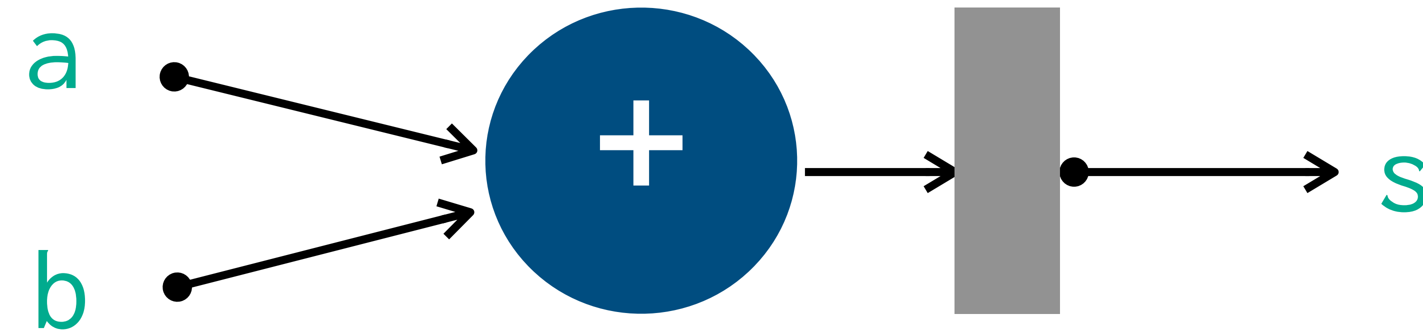
```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
+ step();
+ fork();
  assert_eq(DUT.s, s);
  step();
}
```

A naïve attempt at a pipelined adder

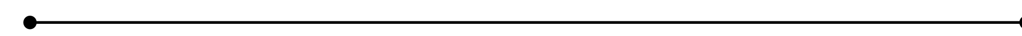



```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  step();
  fork();
  assert_eq(DUT.s, s);
  step();
}
```

A **naïve** attempt at a pipelined adder

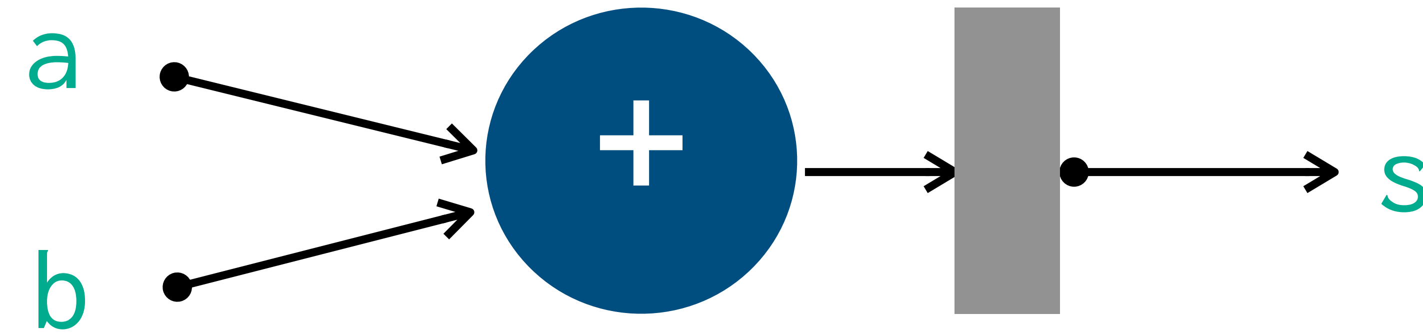



```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  step();
  fork();
  assert_eq(DUT.s, s);
  step();
}
```



`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();
}
```

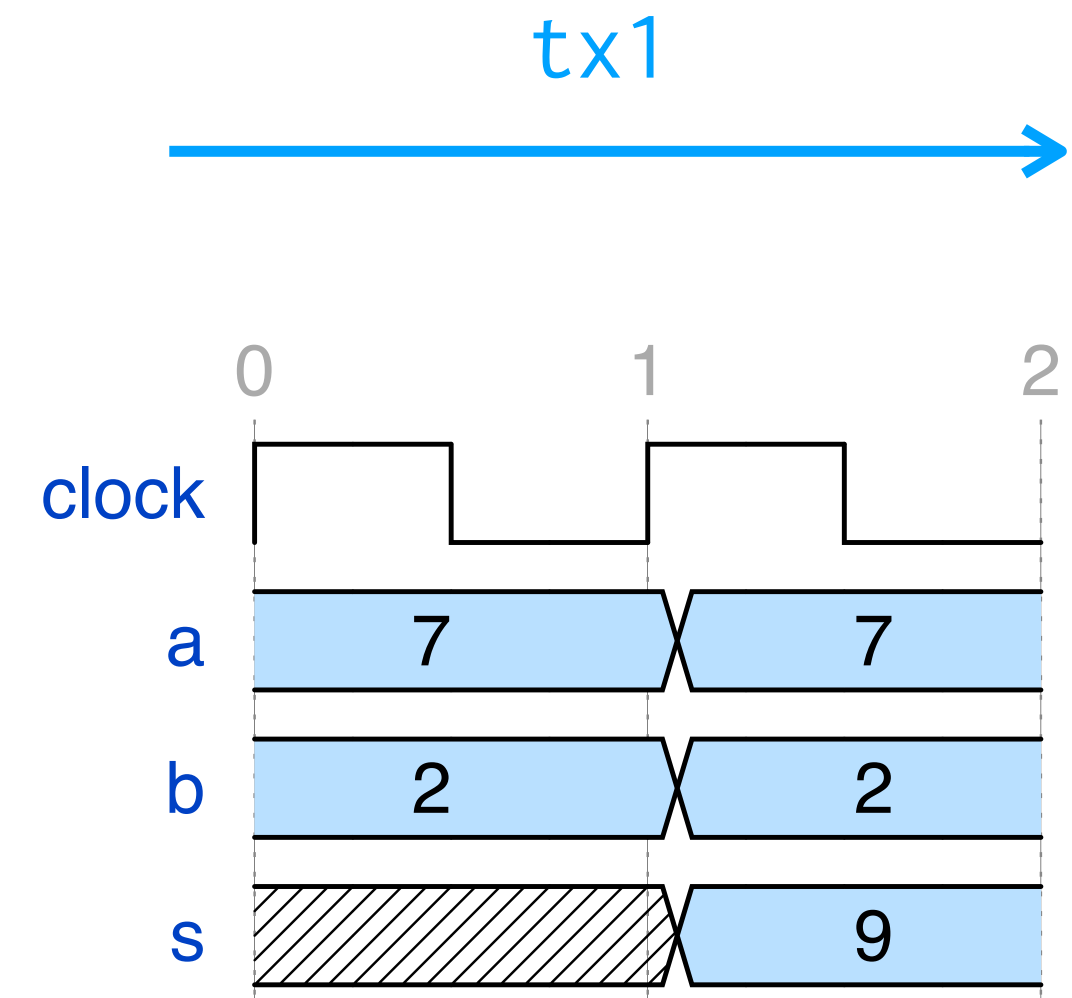
2nd `step()` for a total latency of 2 cycles

A naïve attempt at a pipelined adder

tx1: add(7, 2, 9)

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();
}
```

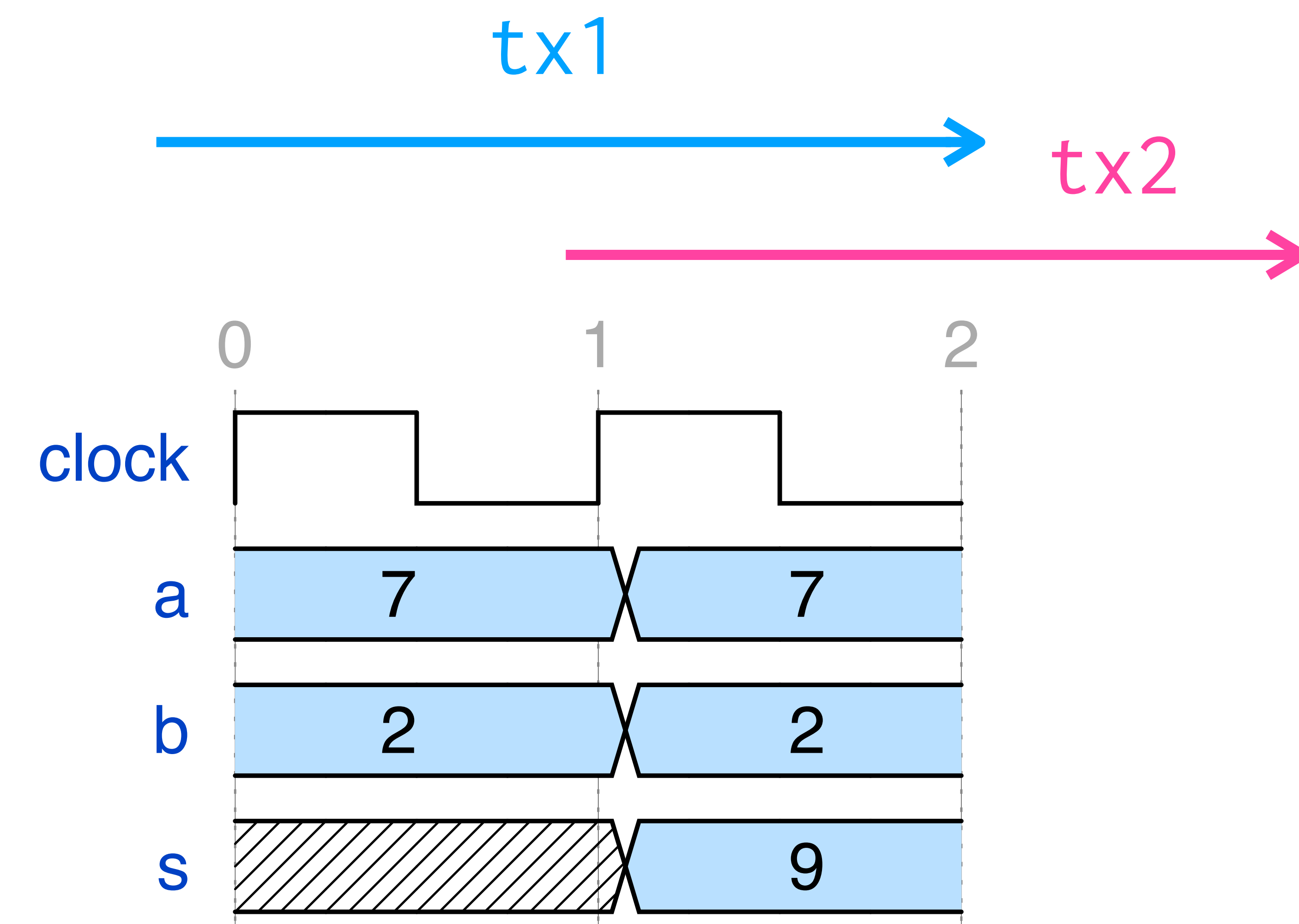


A **naïve** attempt at a pipelined adder

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();
}
```

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



A naïve attempt at a pipelined adder

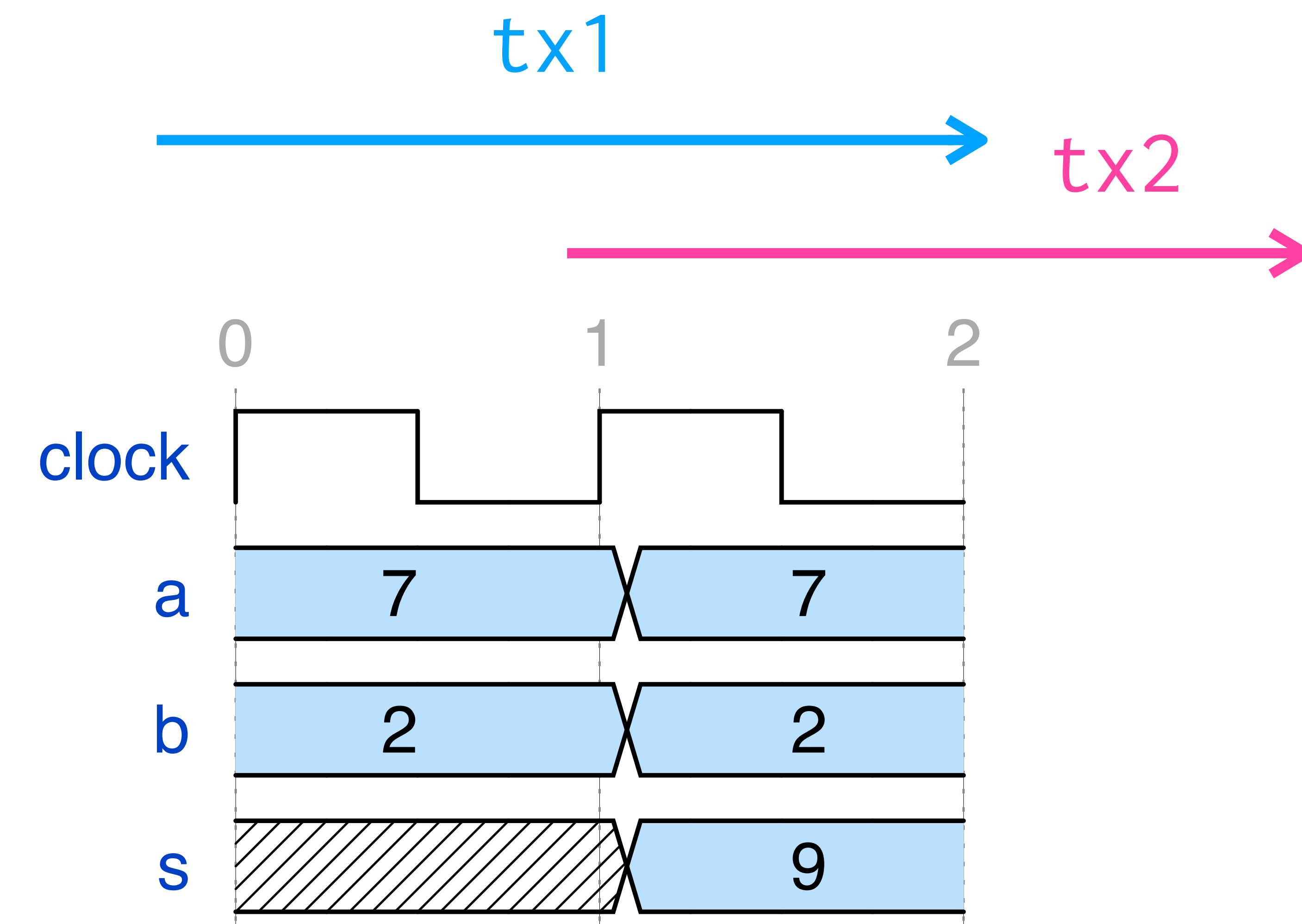
tx1: add(7, 2, 9)

tx2: add(2, 4, 6)

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!



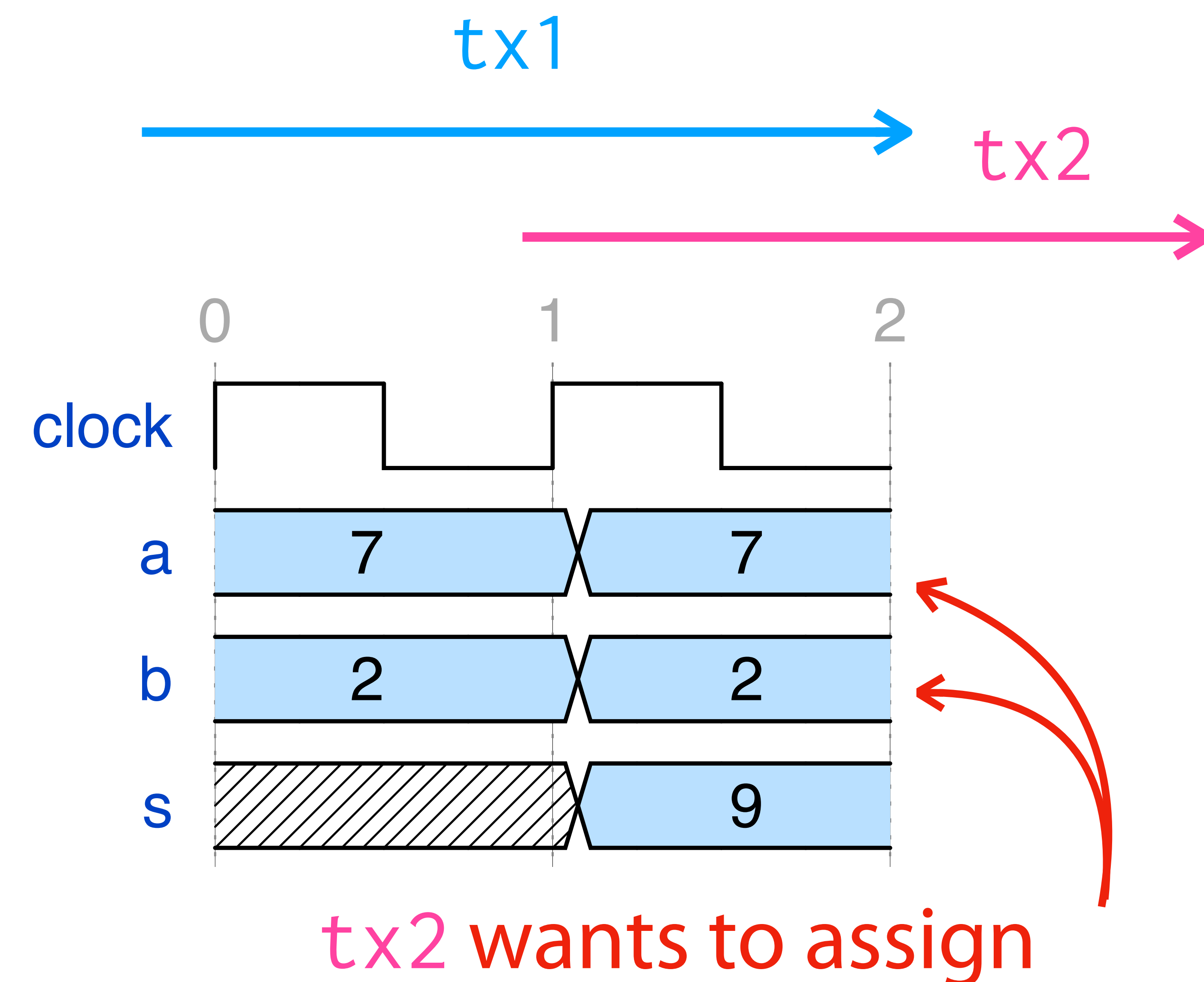
A naïve attempt at a pipelined adder

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)

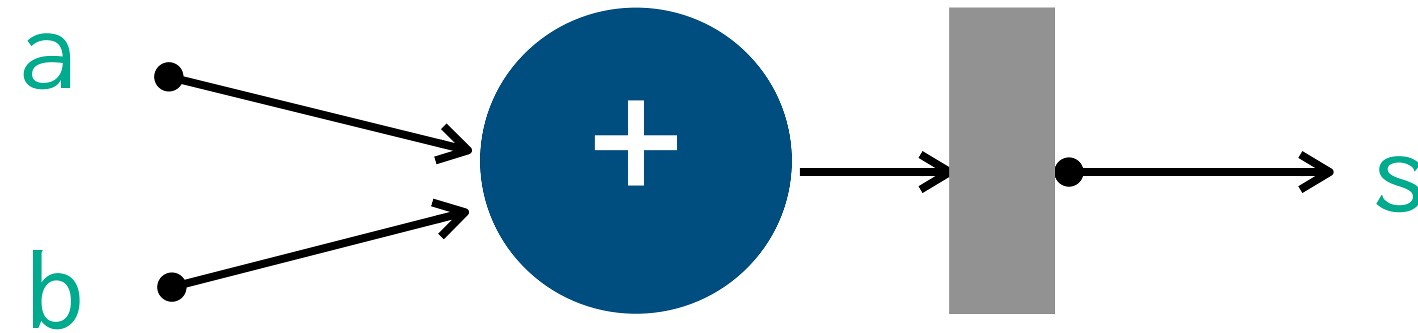


tx2 wants to assign

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

here but is unable to do so!

A pipelined adder, take two



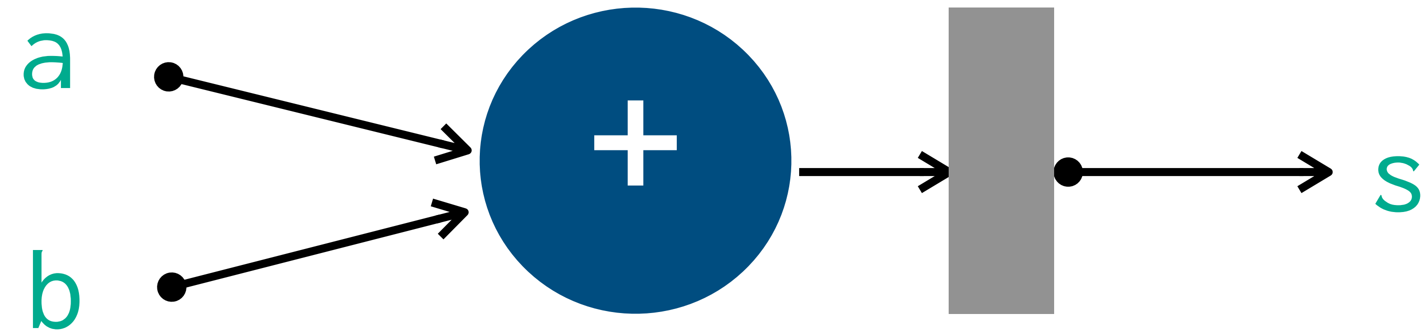
Combinational

```
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

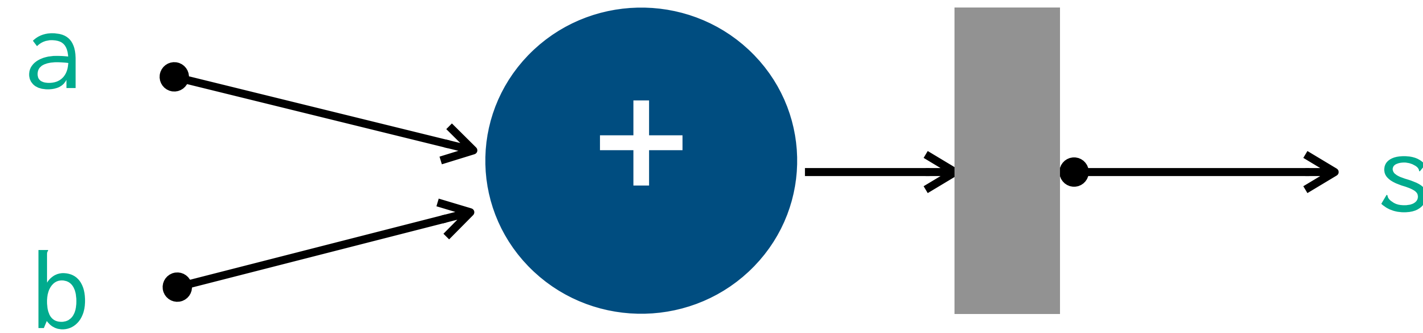
```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;



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

Pipelined

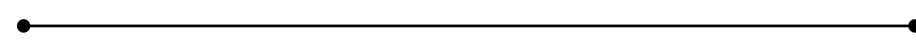
```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
+ step();
+ DUT.a := X; DUT.b := X;
+ fork();
  assert_eq(DUT.s, s);
  step();
}
```

← new!

A pipelined adder

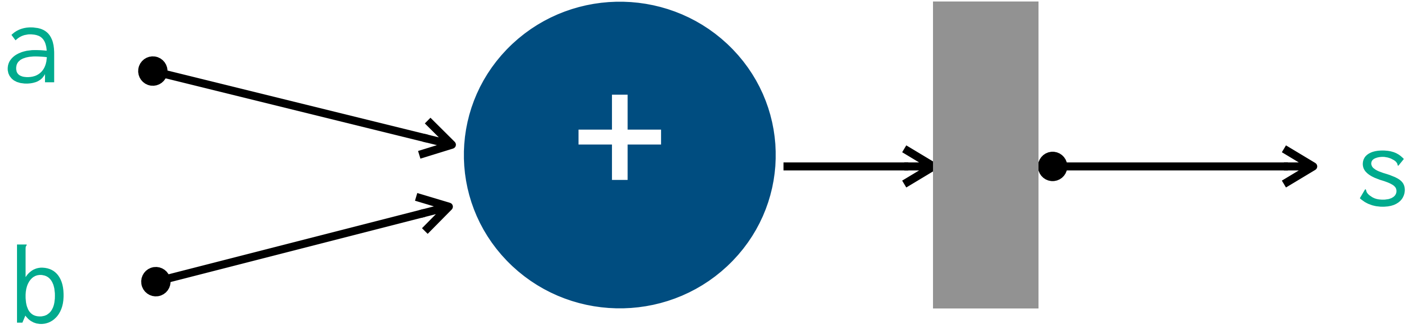



```
prot add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b;
  step();
  DUT.a := X; DUT.b := X;
  fork();
  assert_eq(DUT.s, s);
  step();
}
```



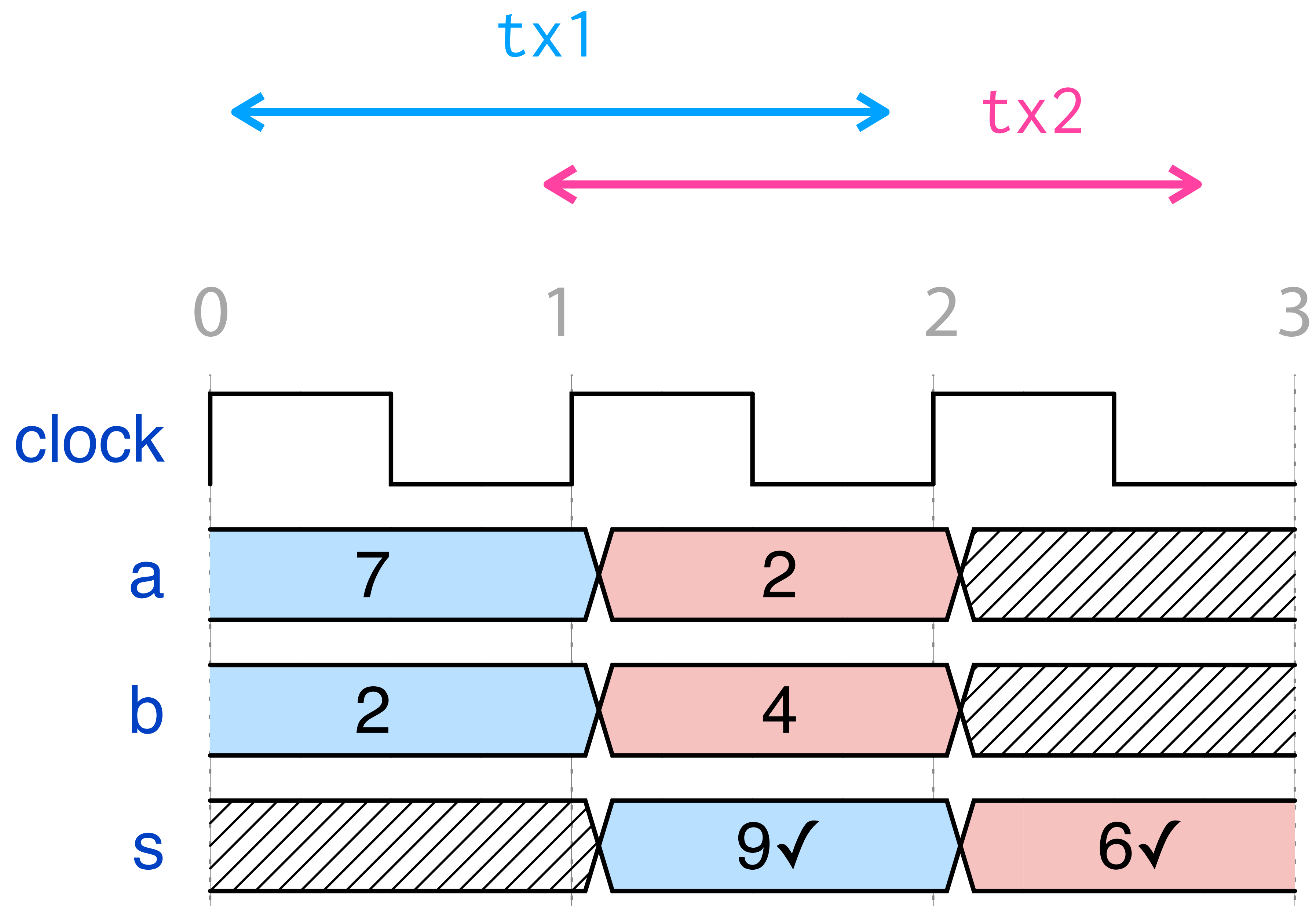
`DontCare` assignments: release constraints on ports
(Value on ports `DUT.a` & `DUT.b` no longer matter)

A pipelined adder




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

```
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();
}
```



Second example:
A Ready-Valid Handshake in Paso

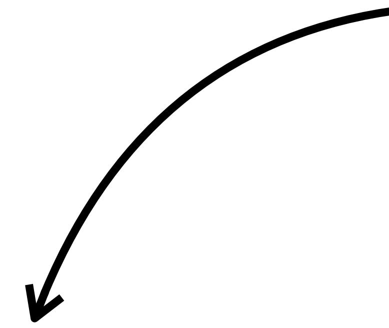
A Ready-Valid Handshake in Paso

A Ready-Valid Handshake in Paso

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




}
```

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

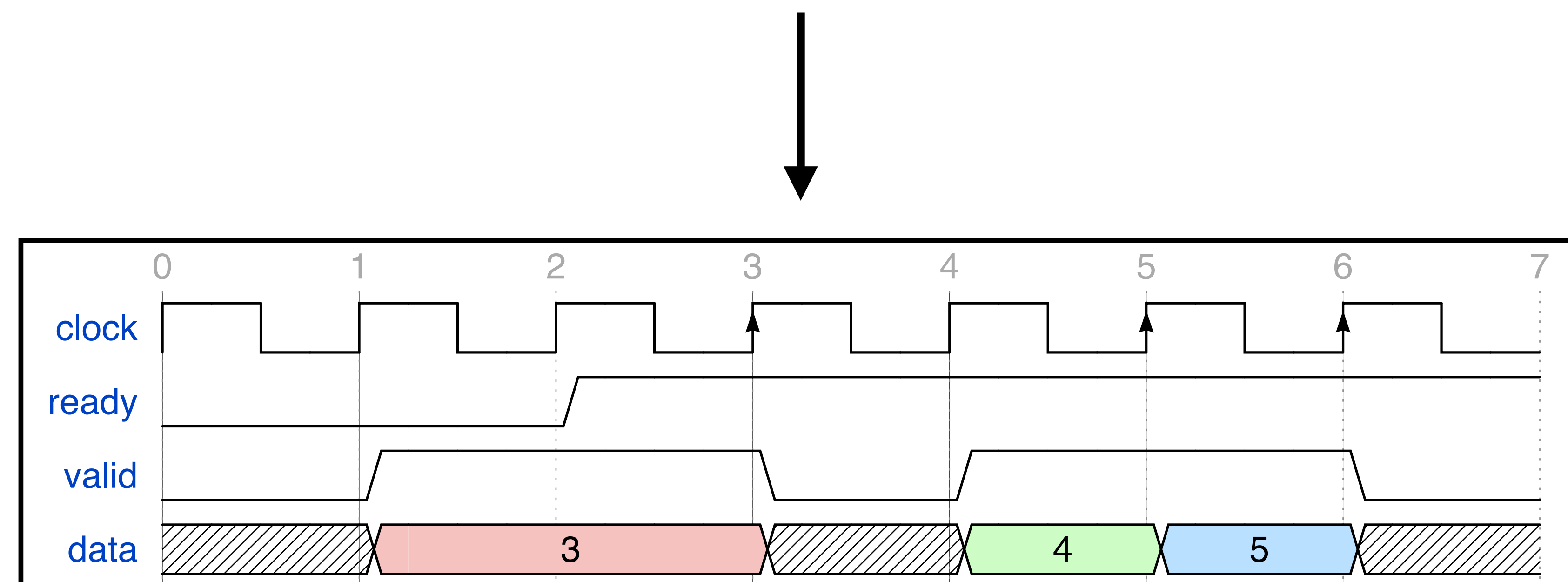
Simple imperative semantics, just like software!

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

What can we do with one single protocol?

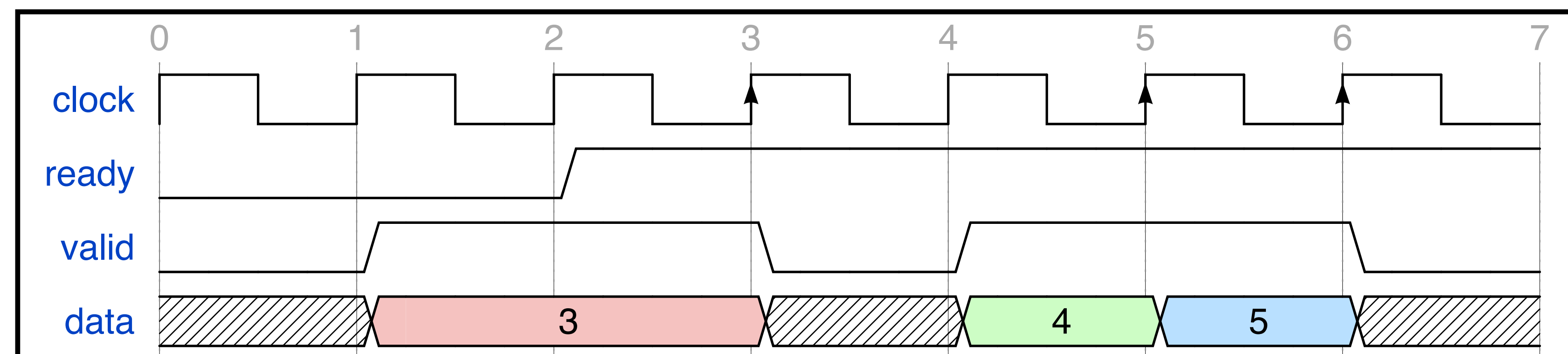
What can we do with one single protocol?

1. Run concrete tests
(drive hardware modules)

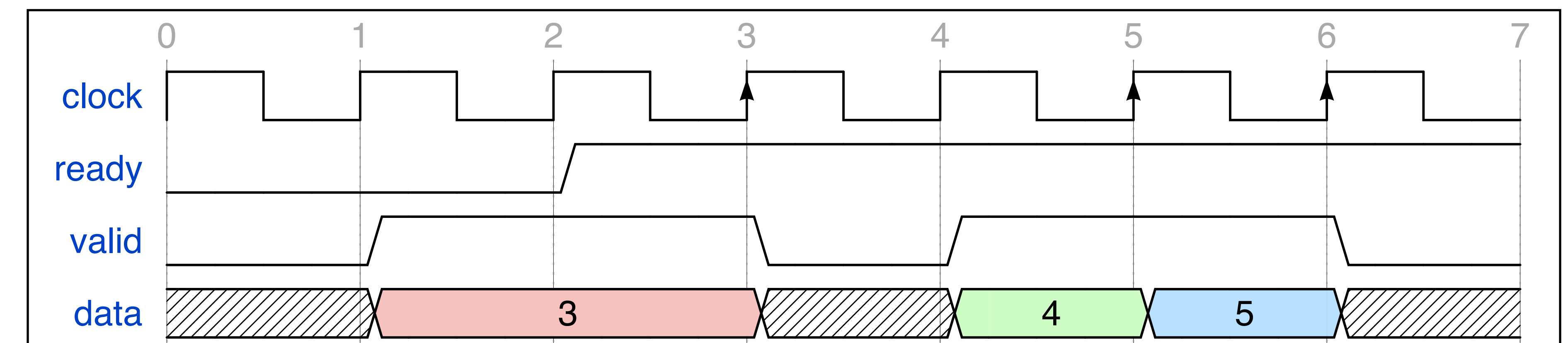


What can we do with one single protocol?

1. Run concrete tests (drive hardware modules)



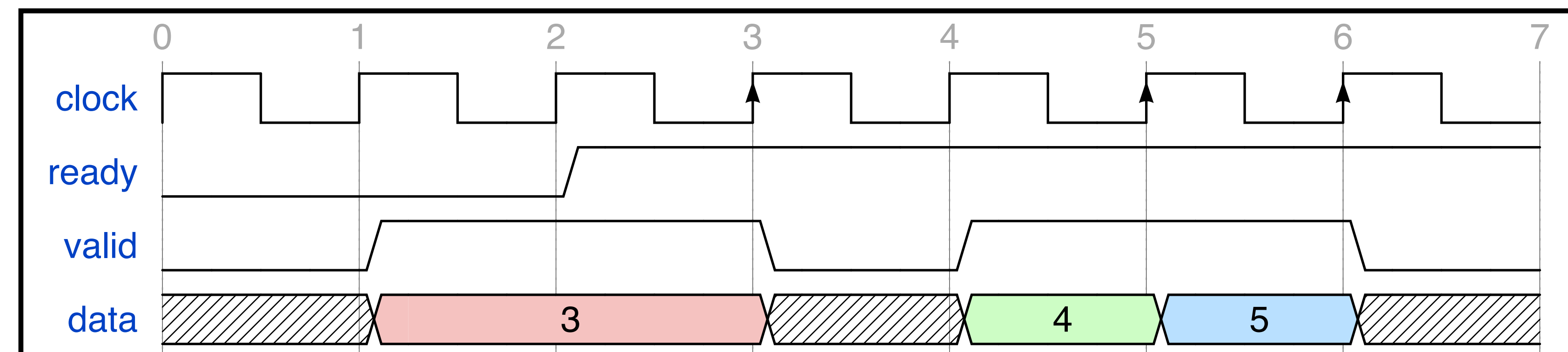
2. Infer transactions from waveforms



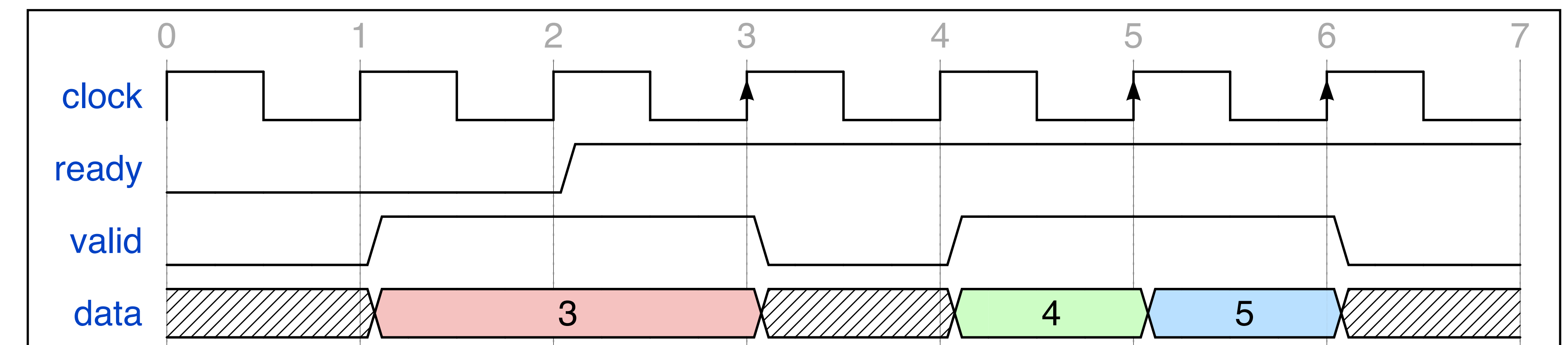

```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

Our DSL: 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)
- a ***reconstructor*** (infers transactions from signals)

Interpreter

Paso protocol spec

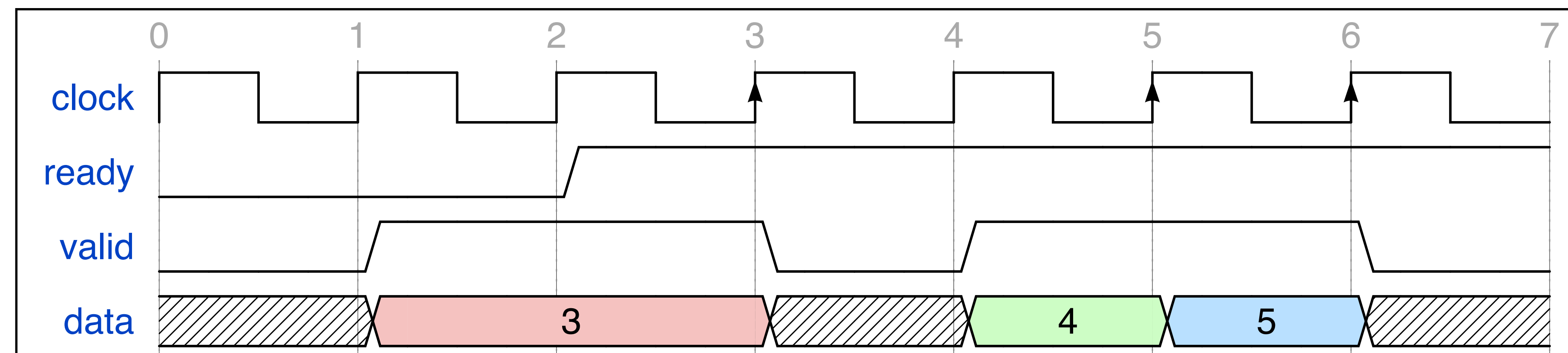
Hardware module implementation

Transactions to execute

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

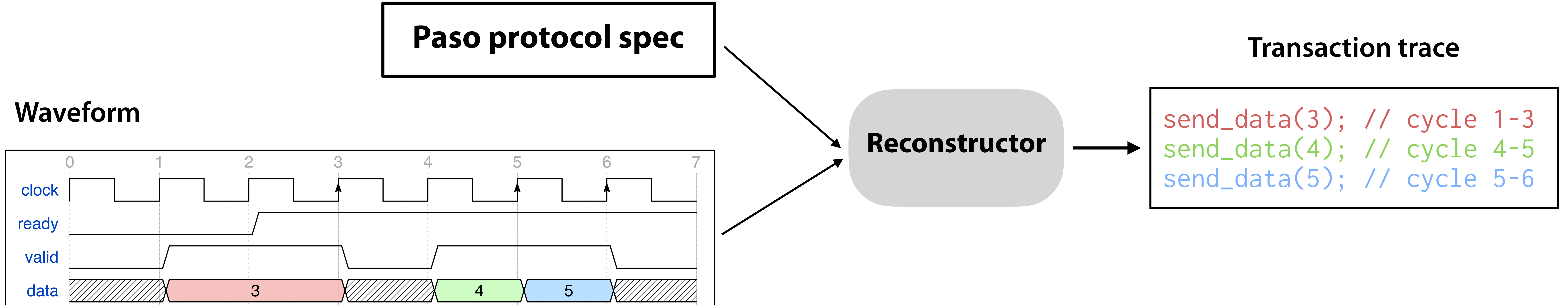
Interpreter

Waveform



Reconstructor

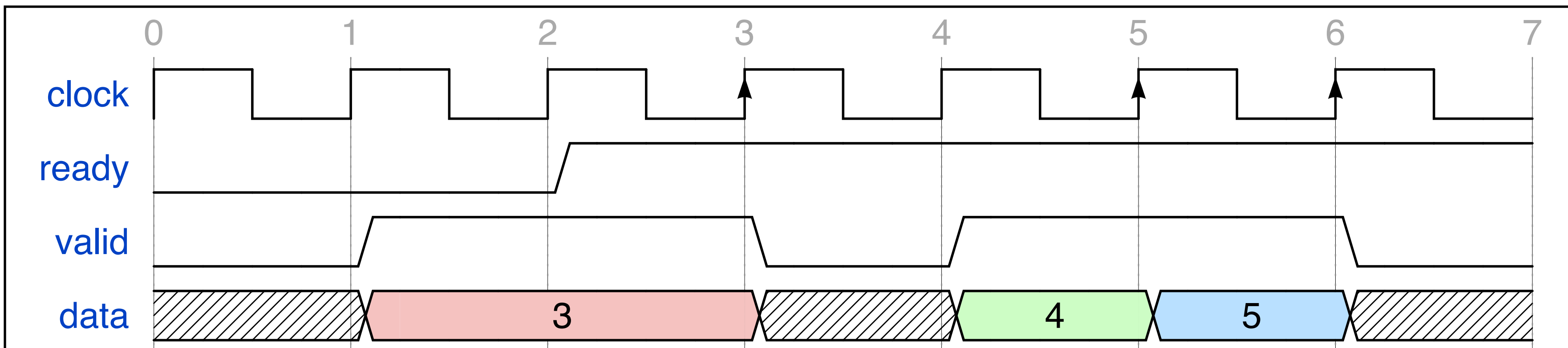
(the cool part!)

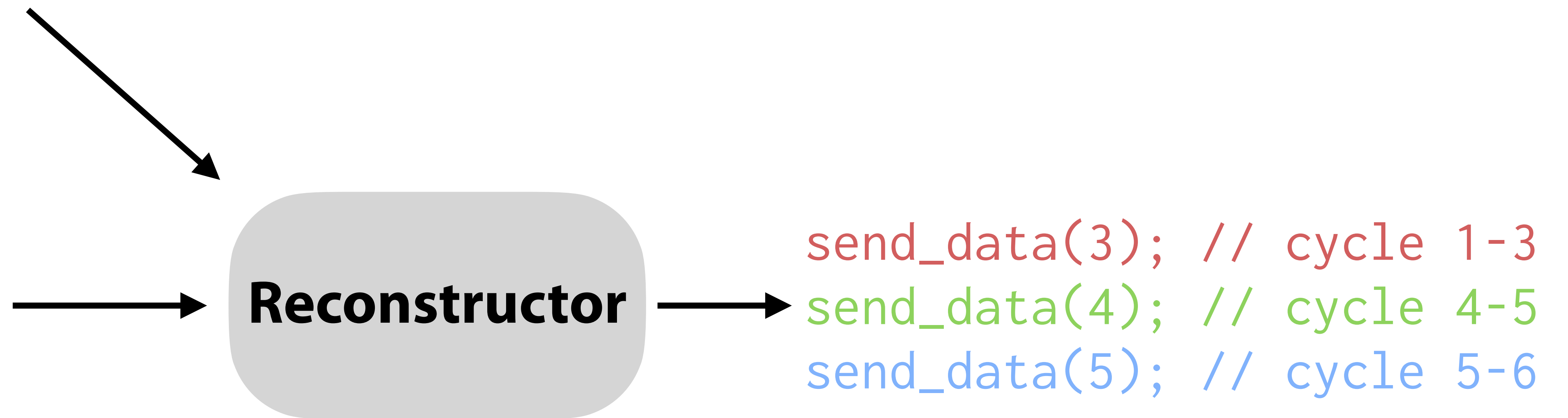


Applying the Reconstructor on the Ready-Valid Example

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) { … }
```



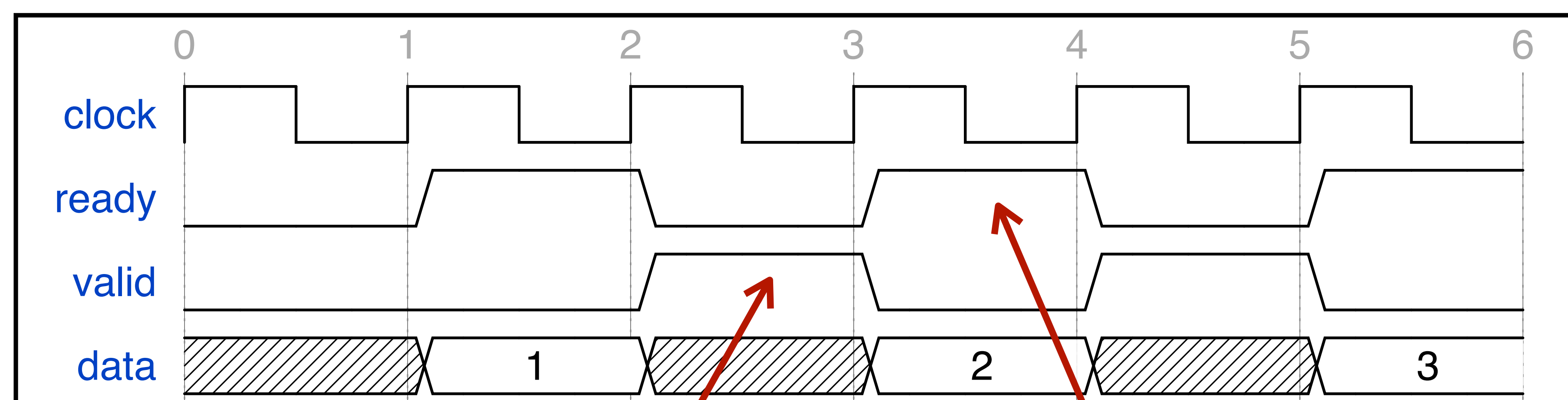


<u>Abstraction level raised from signals to transactions!</u>

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)

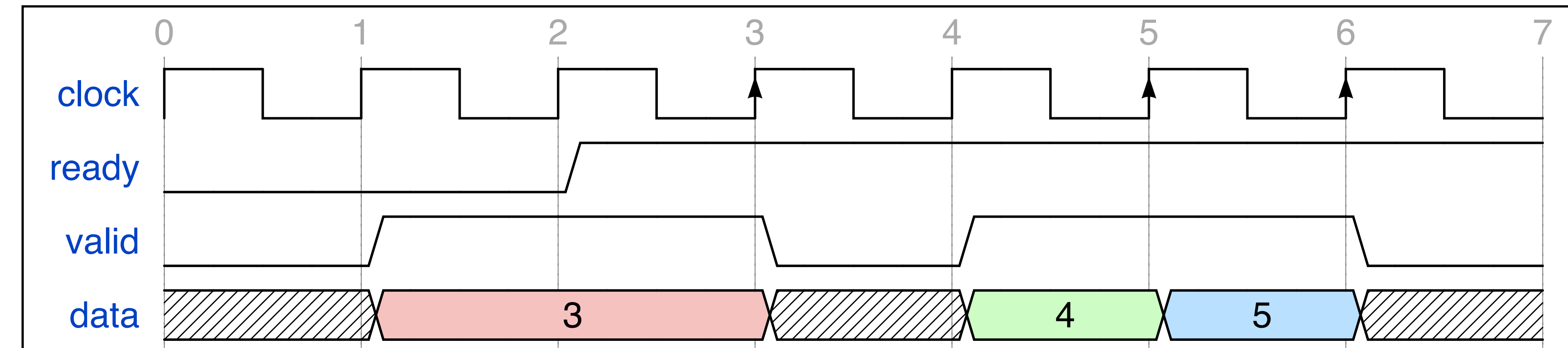
Round-trip property

Interpreter

User-supplied transactions

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

Waveform



Reconstructor

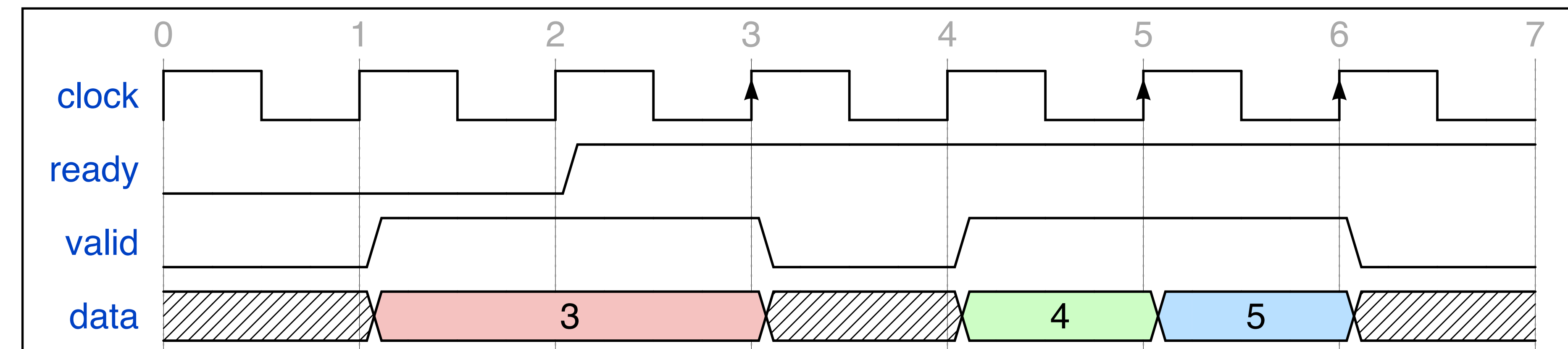
Round-trip property

Interpreter

Waveform

User-supplied transactions

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



Reconstructor

Designing the reconstructor involves "reversing" Paso's semantics!

How does the reconstructor work?

Assignments as Constraints

```
foo := 0
```

means

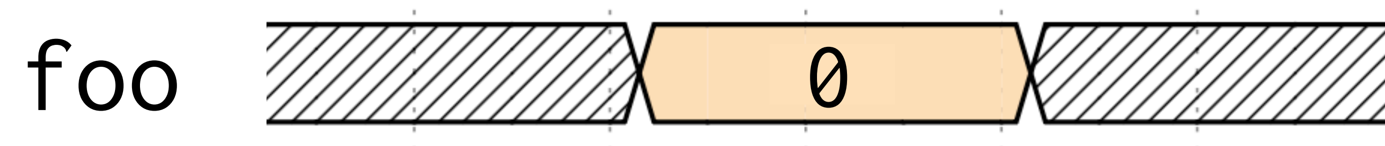
“Check if the current waveform value of `foo` is equal to `0`”

Assignments as Constraints

```
foo := 0
```

means

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

foo 

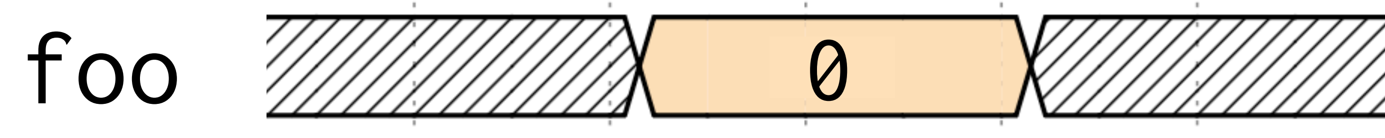
Matches!

Assignments as Constraints

```
foo := 0
```

means

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

foo 

Matches!

foo

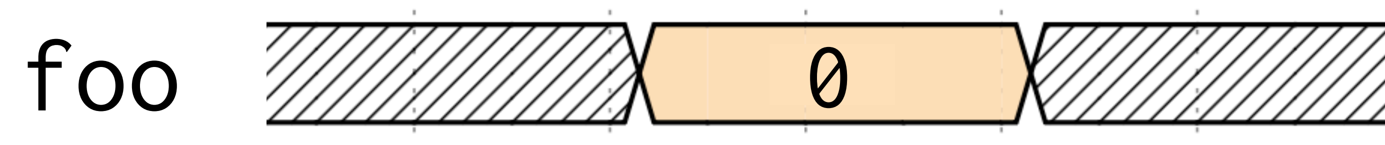
Inconsistent!

Assignments as Constraints

```
foo := 0
```

means

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



Matches!

Inconsistent!

Treat `foo == 0` as a <u>constraint</u> 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

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

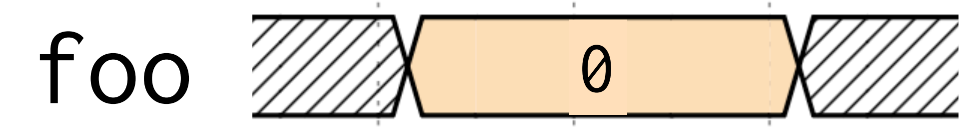
Constraint in reconstructor

`LHS := RHS`

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

`foo := 0`

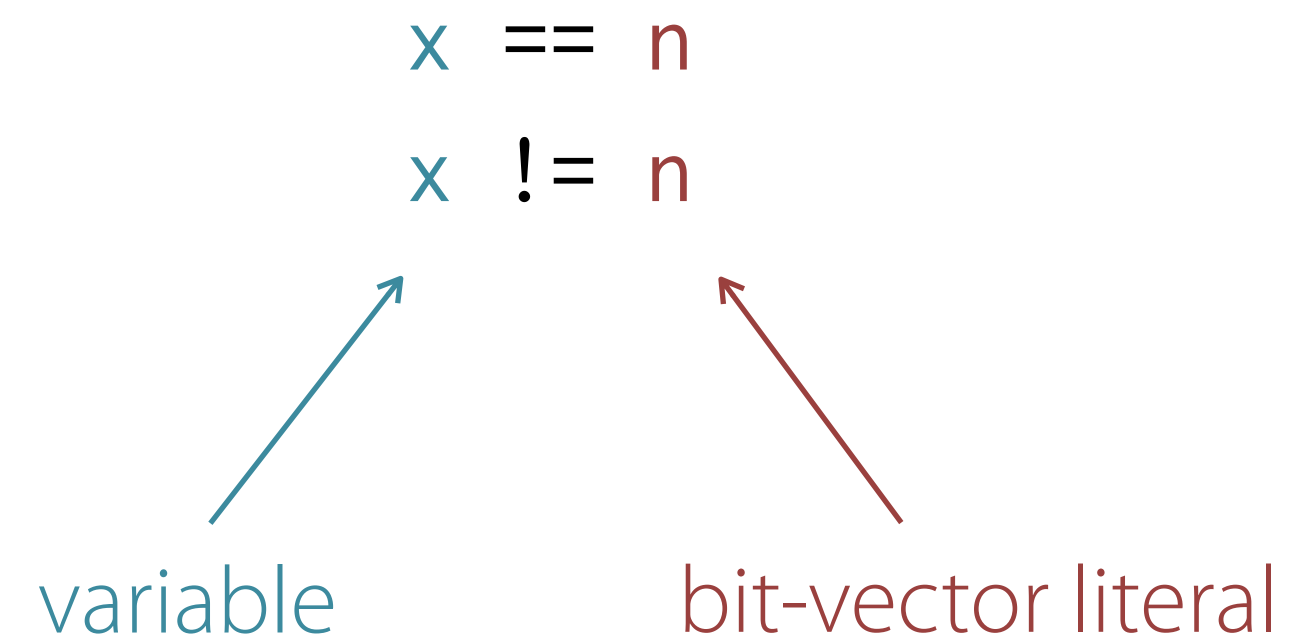
`0 = 0`



`foo := a`

`a = 0`

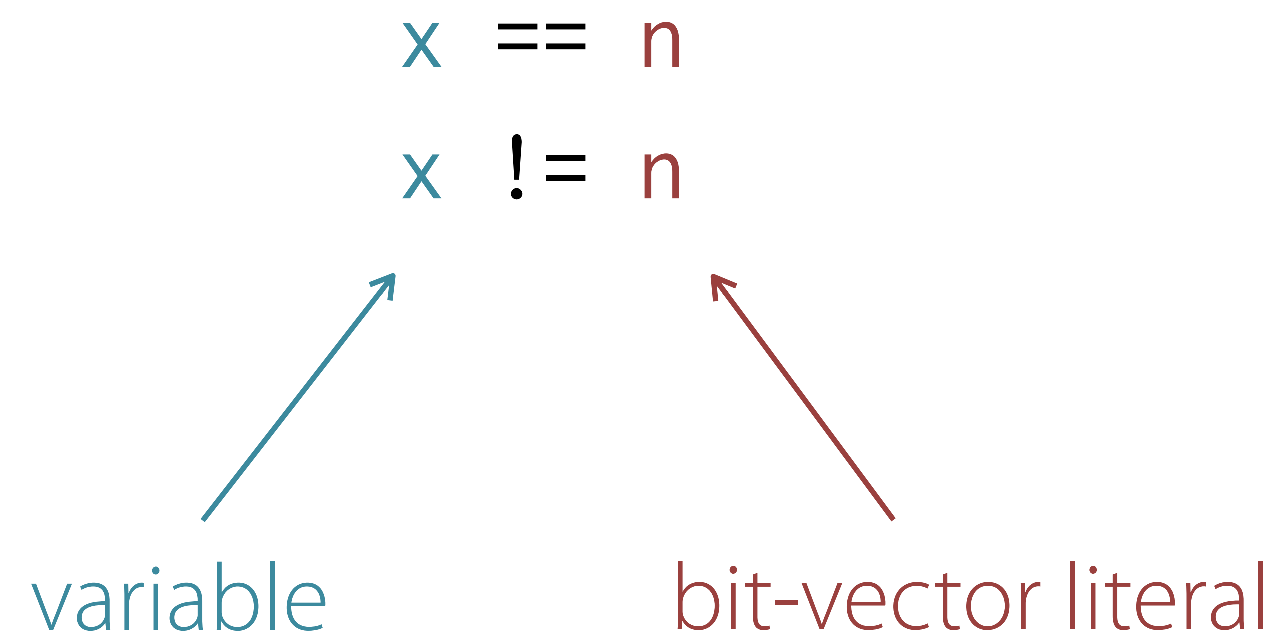
No SMT solvers needed!



No SMT solvers needed!

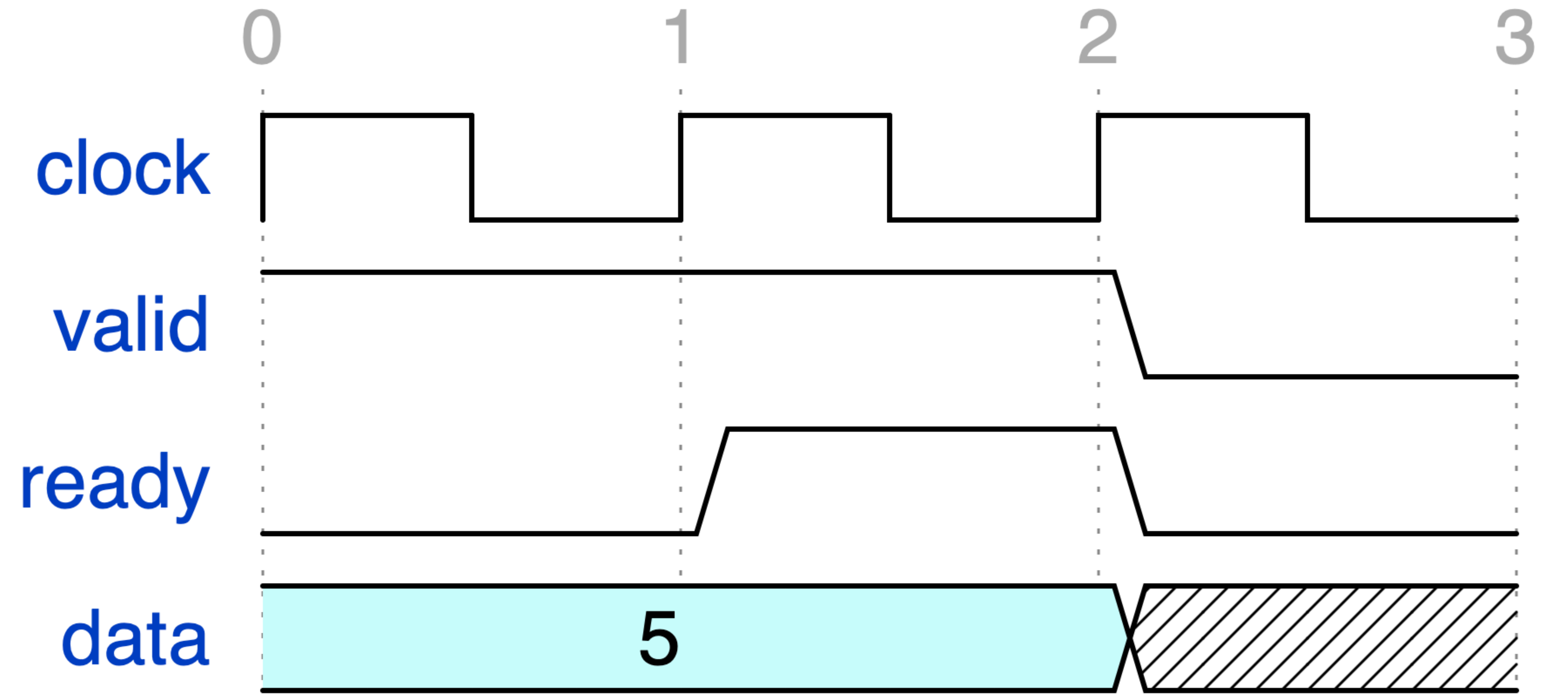
All constraints in our DSL are equality constraints where one side is a constant

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



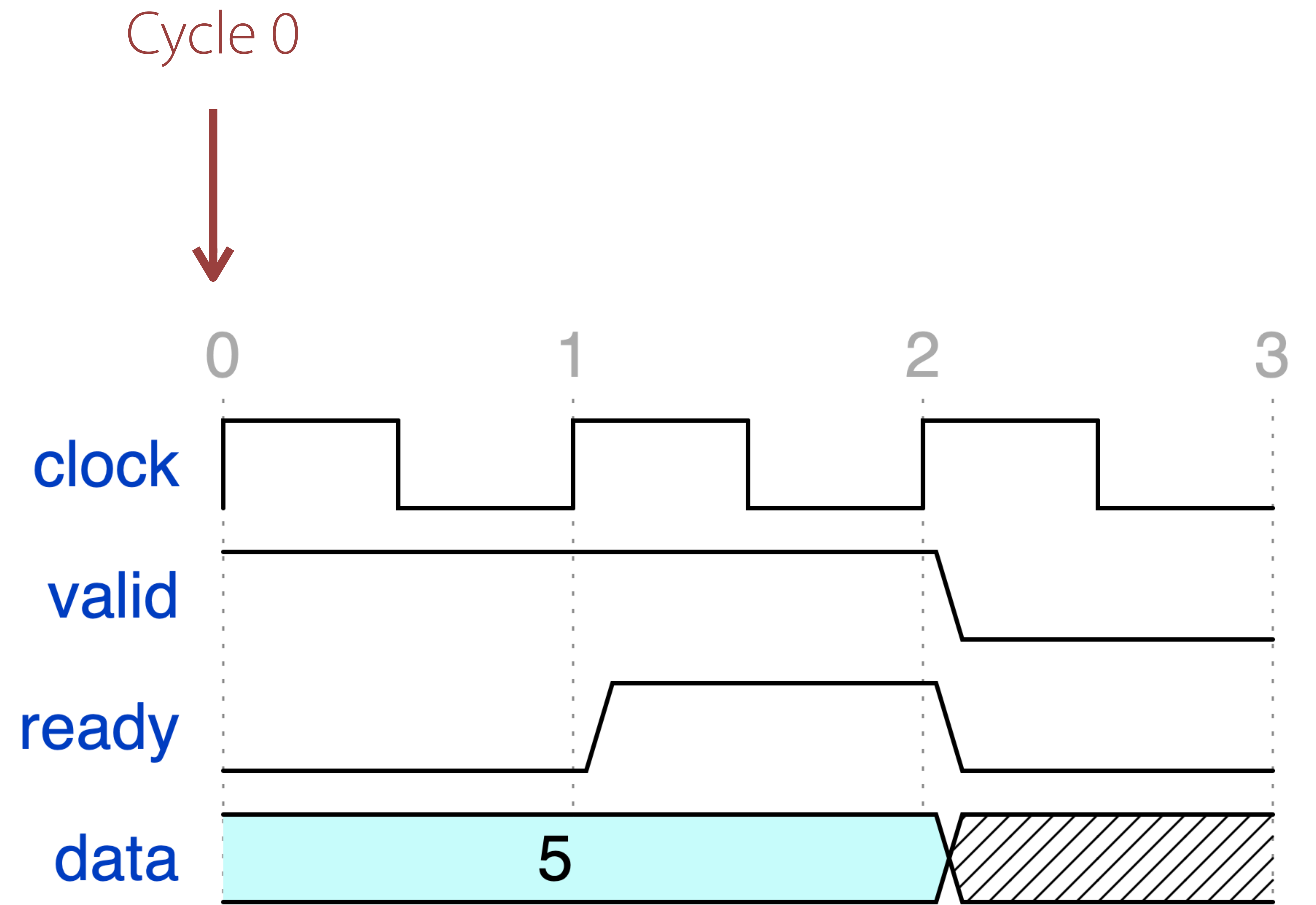
Idea: symbolically execute protocols & compare against waveform

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



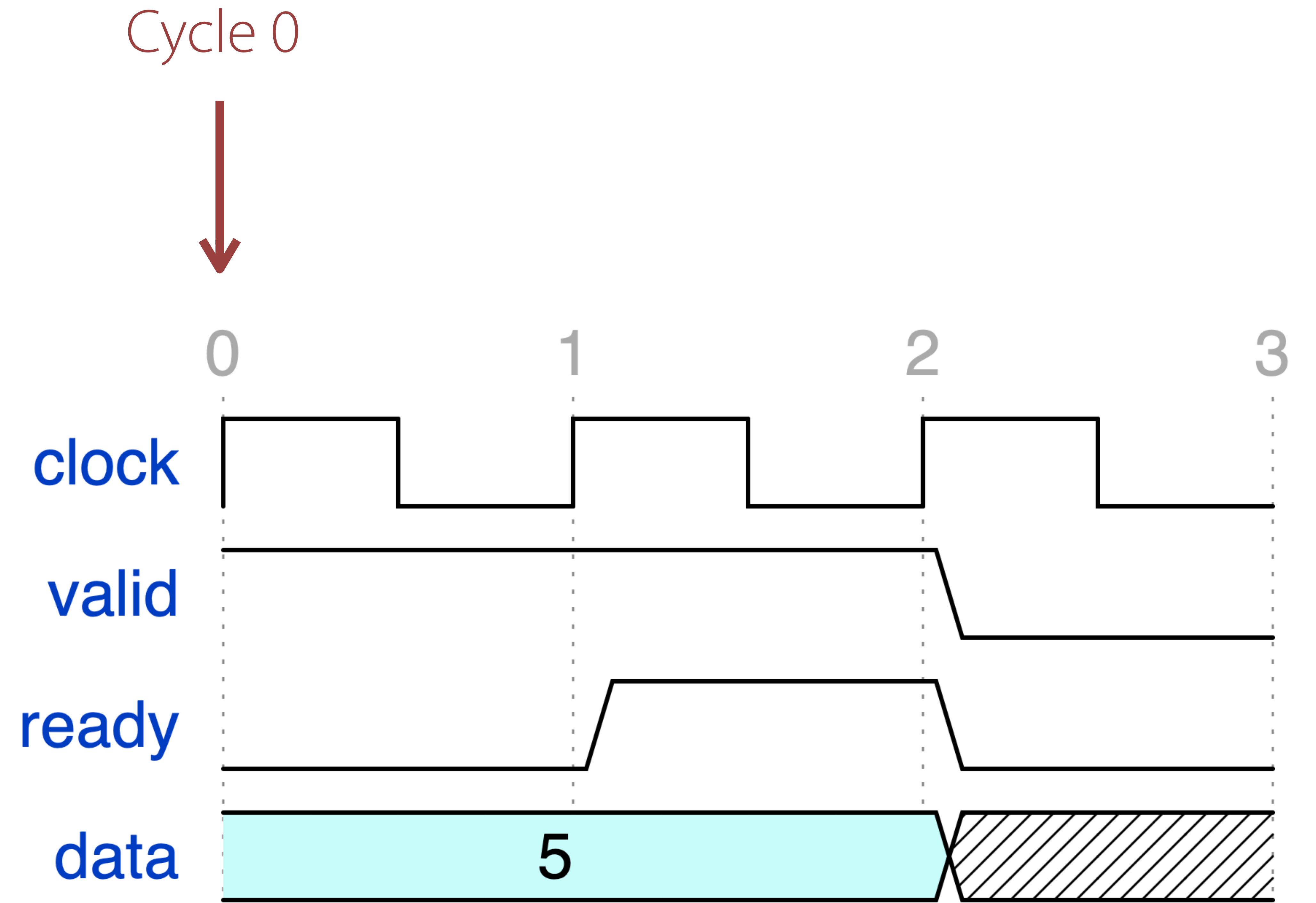
Constraints: ∅

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



Constraints: ∅

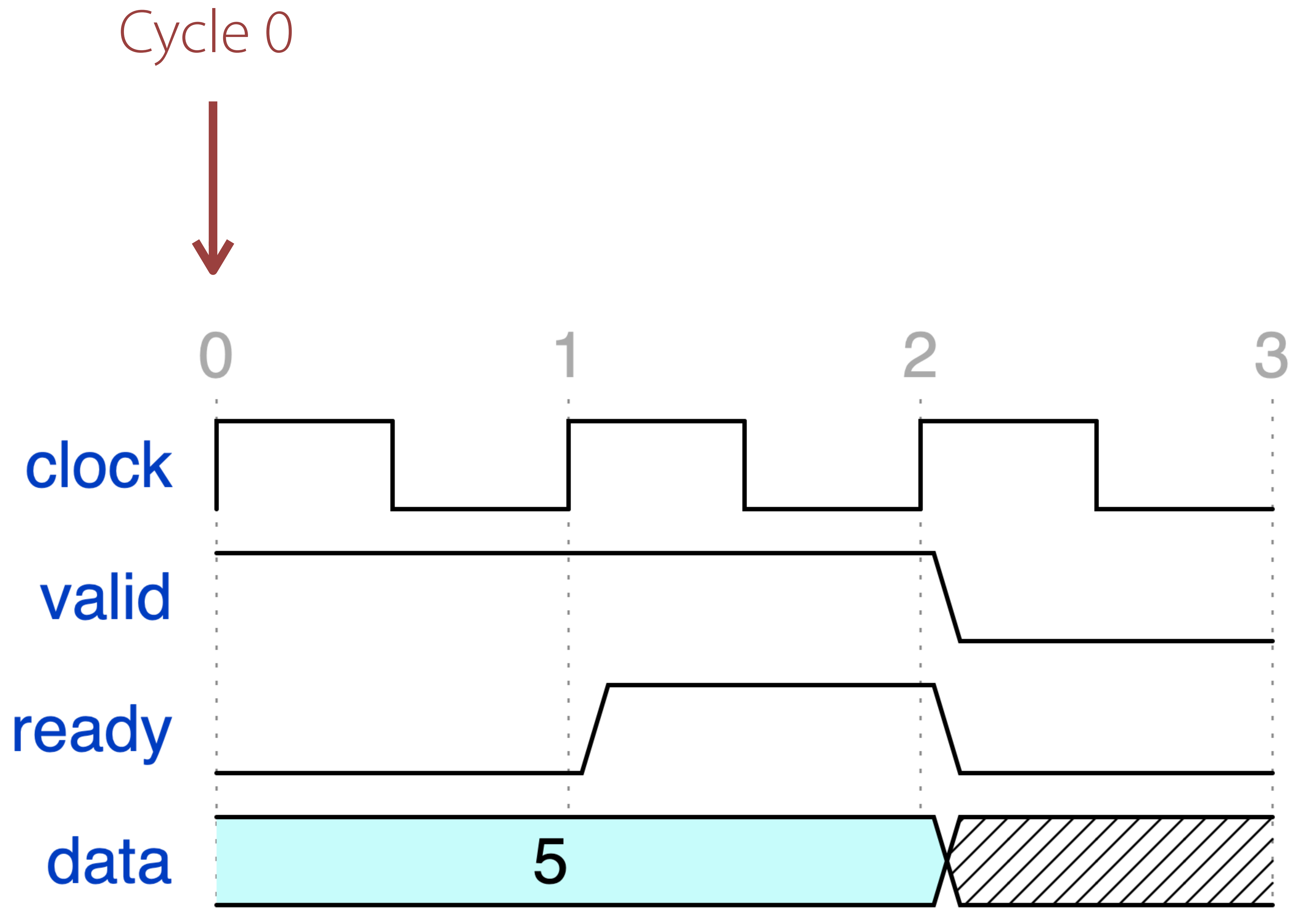
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```



Constraints: ∅

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

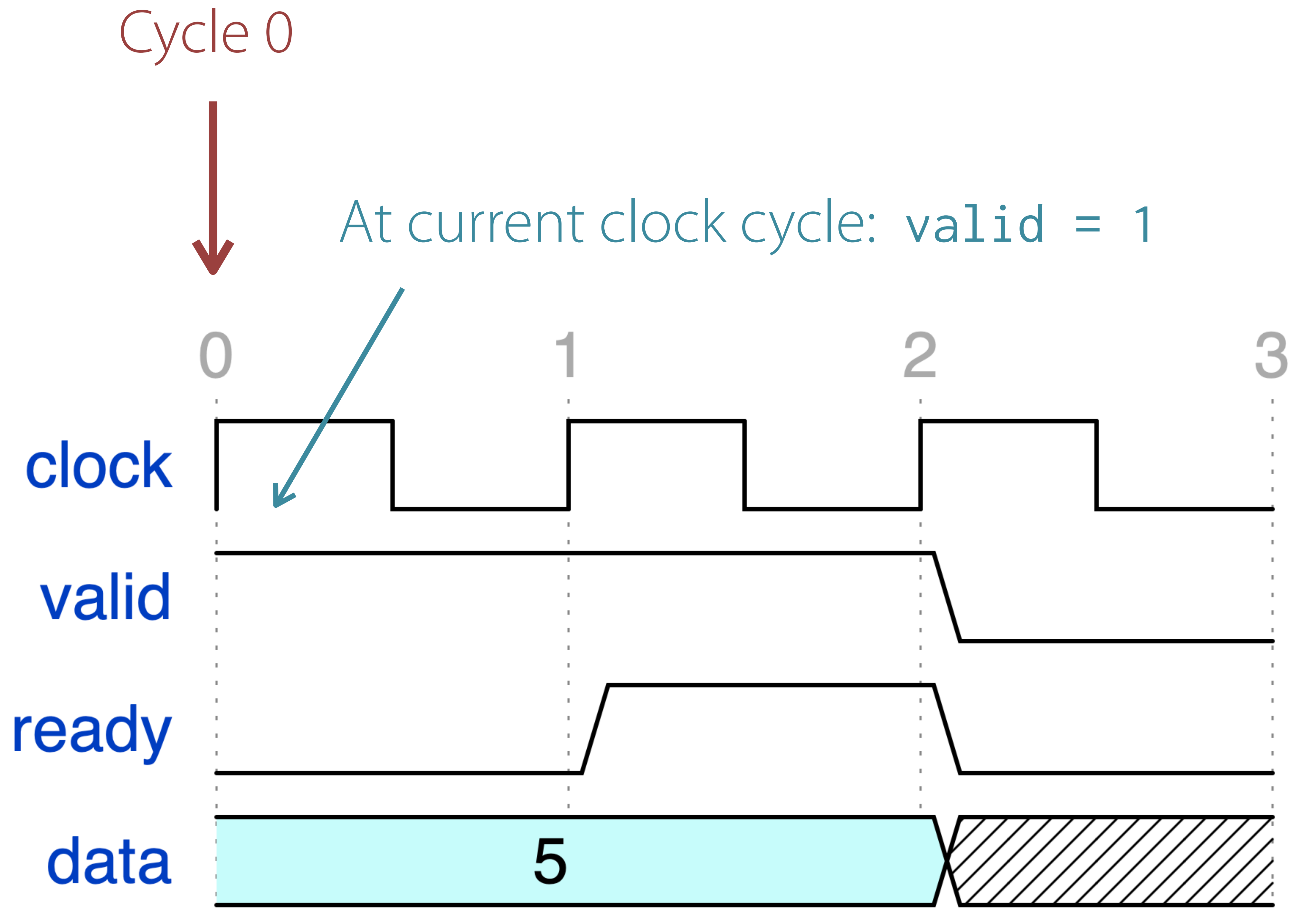
Check if current waveform value of `valid =? 1`



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`



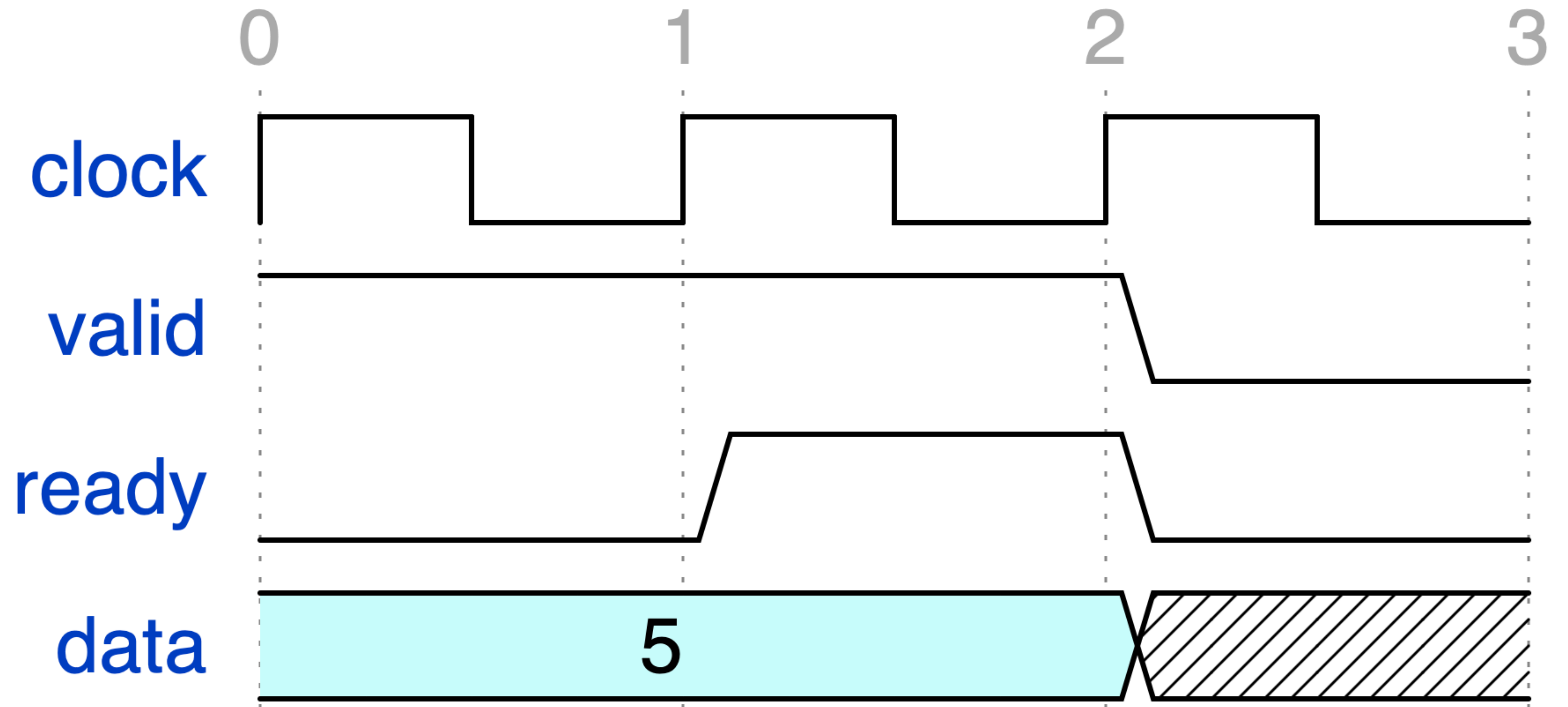
Constraints: { 1 = 1 }

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

Cycle 0

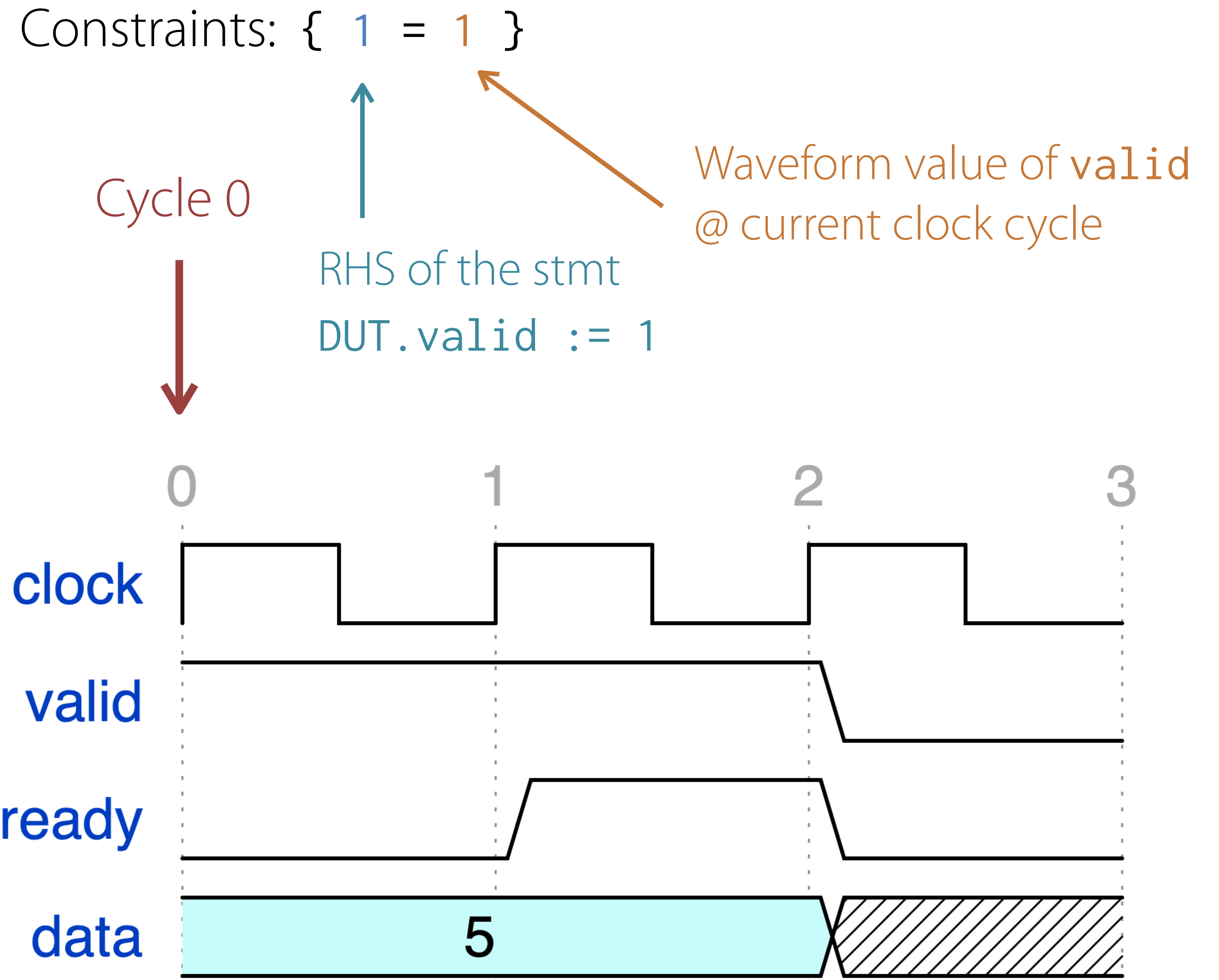
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

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



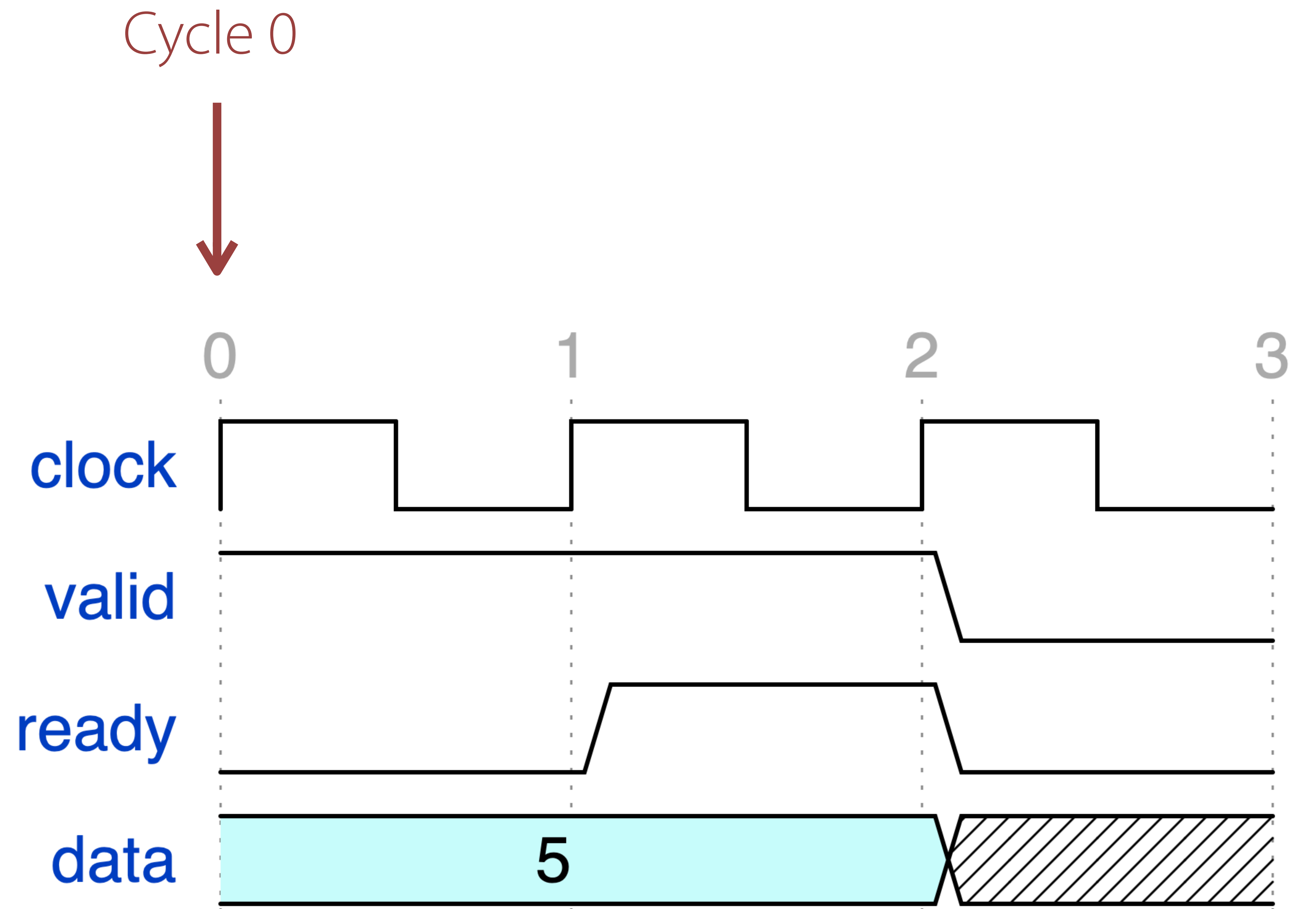
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

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



Constraints: { 1 = 1 }

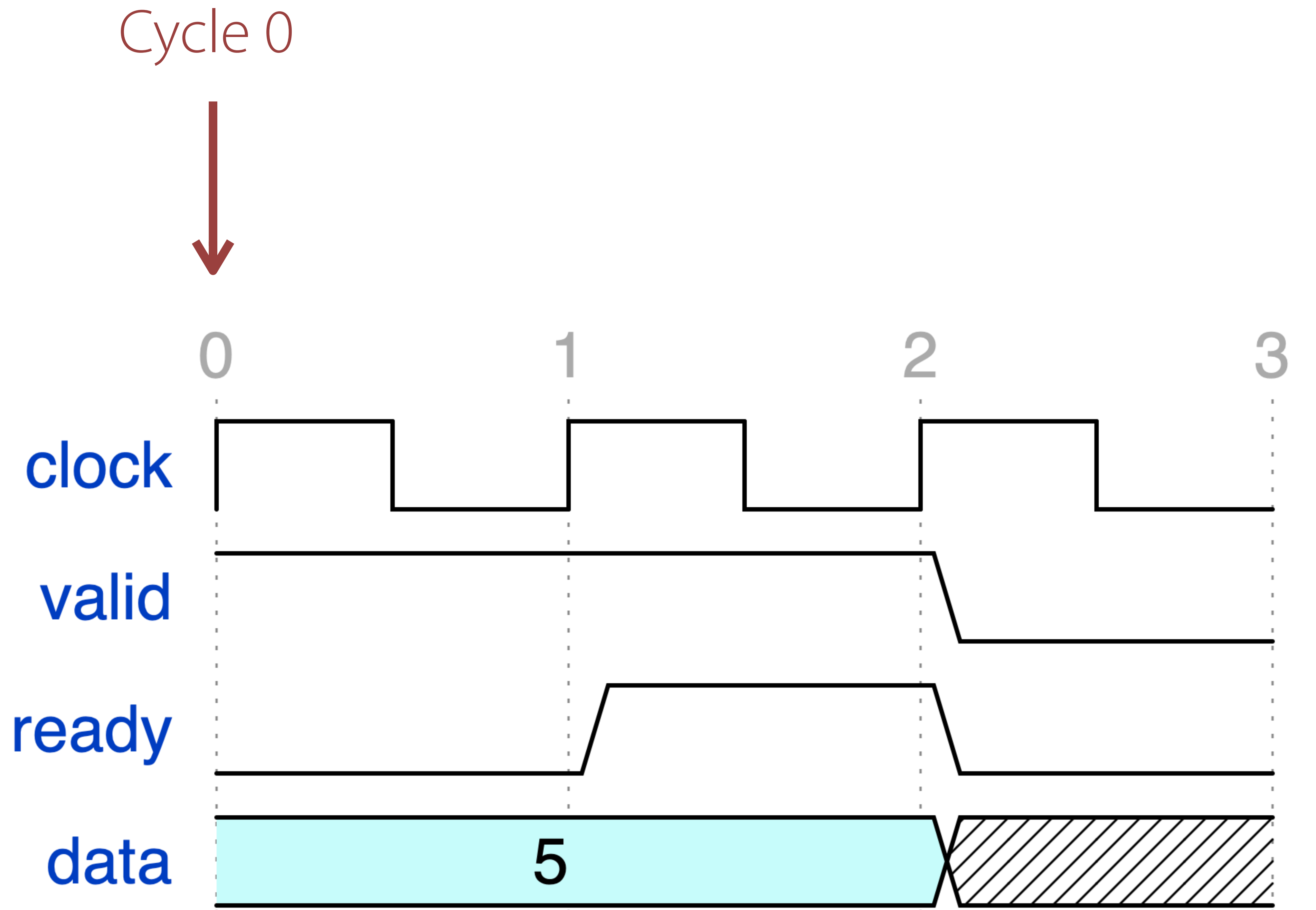
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```



Constraints: { 1 = 1 }

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

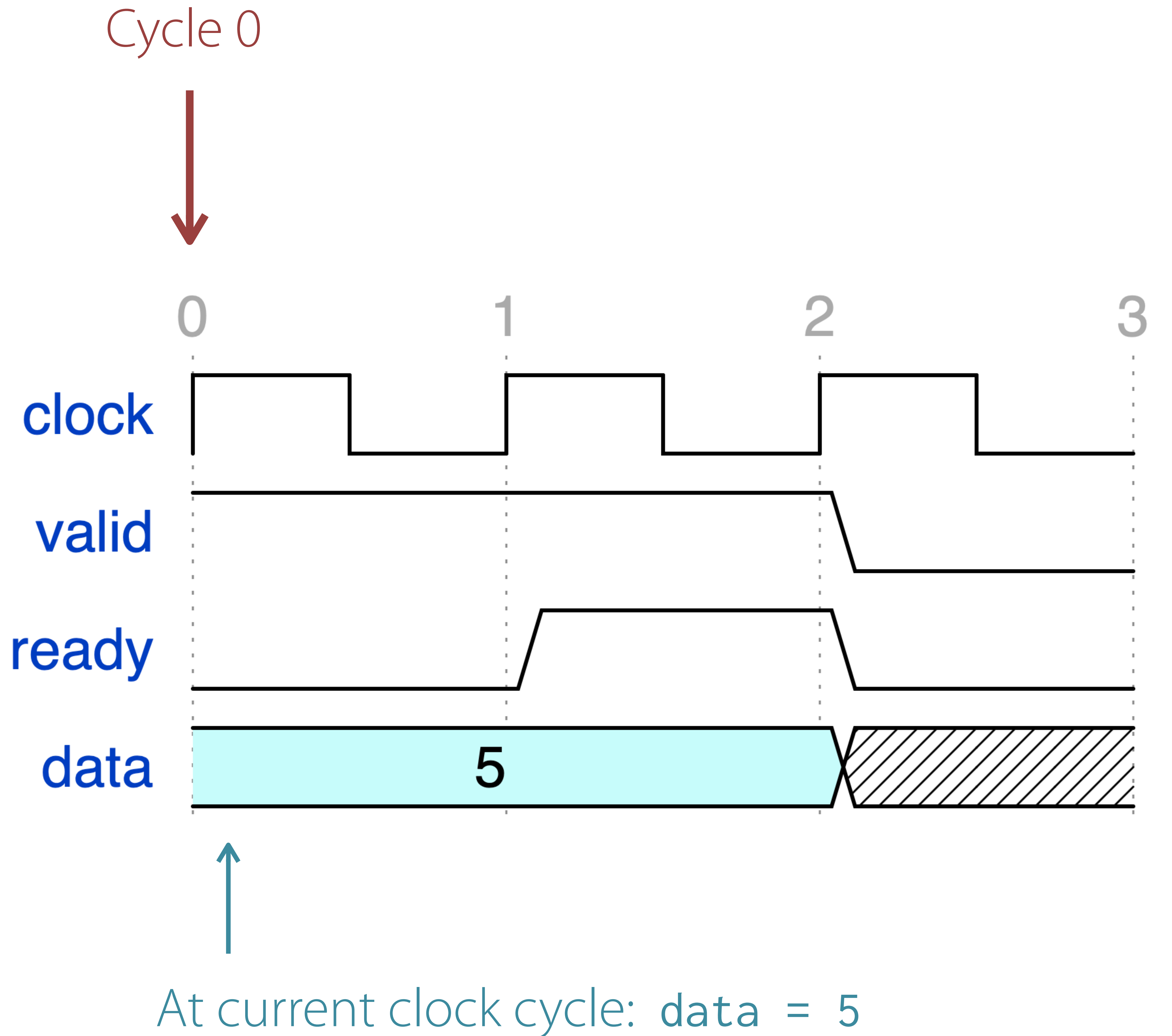
Look up current value of RHS (data) from waveform



Constraints: { 1 = 1 }

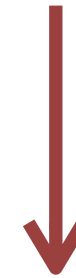
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

Look up current value of RHS (data) from waveform



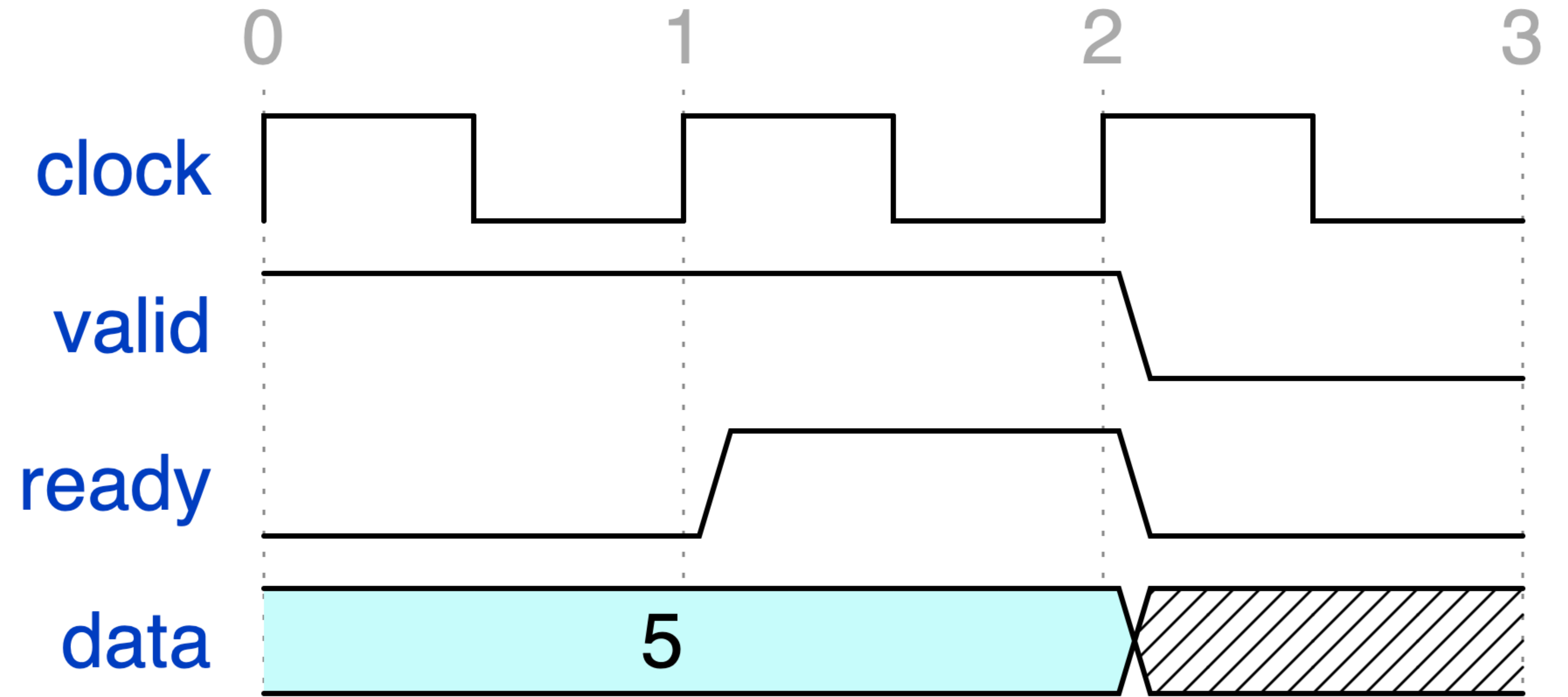
Constraints: { 1 = 1, data = 5 }

Cycle 0



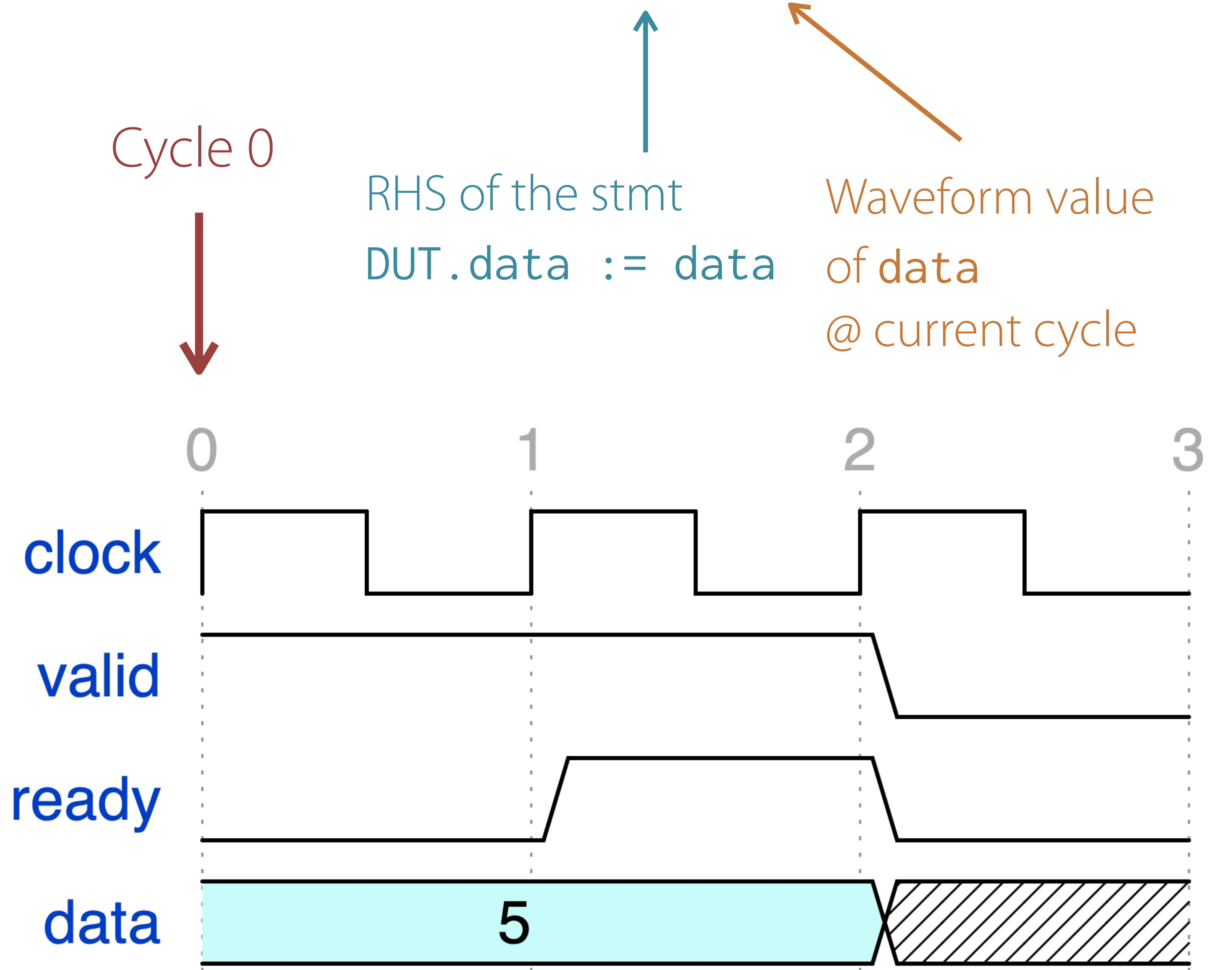
Add a new constraint data = 5

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



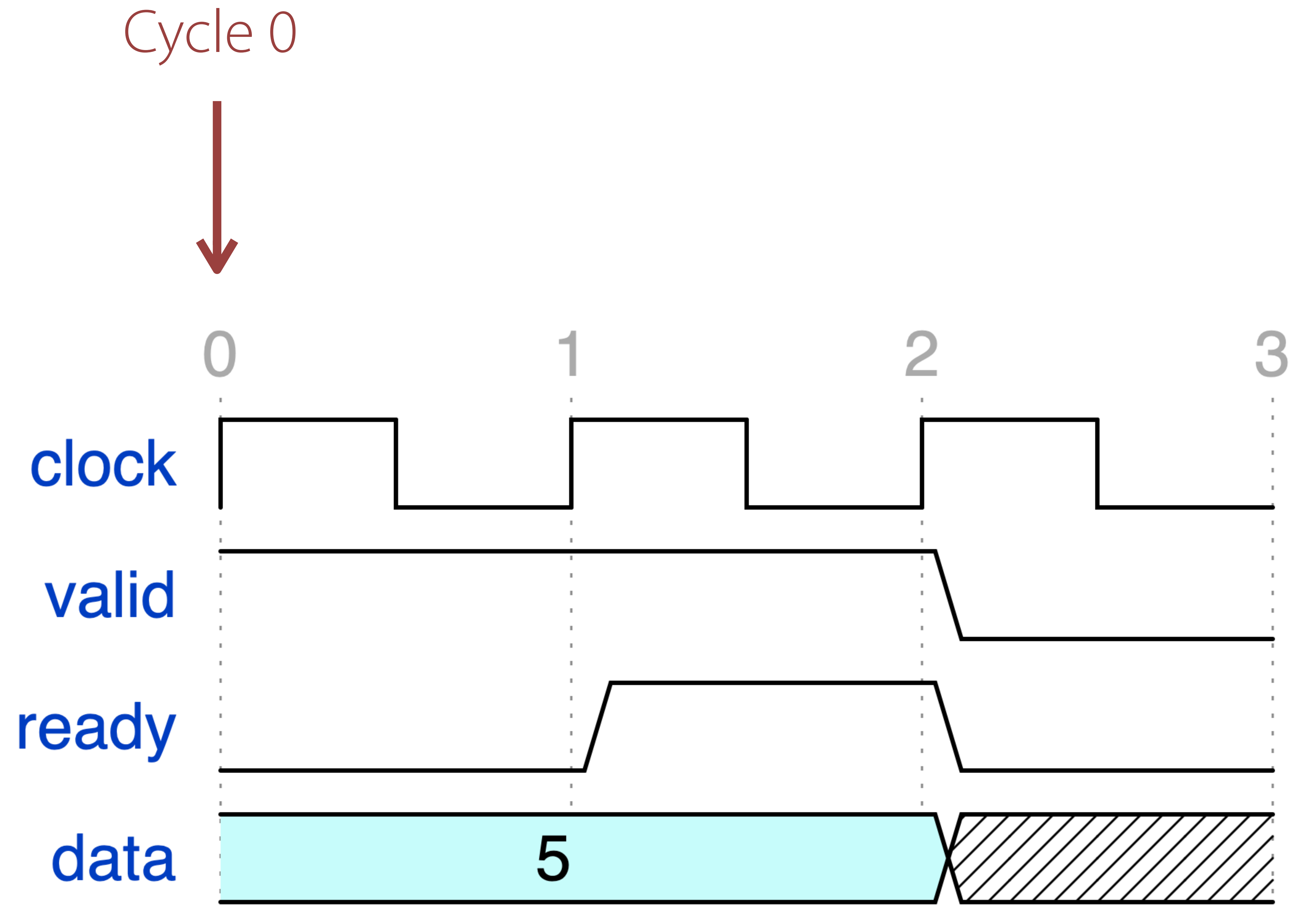
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

Constraints: { 1 = 1, data = 5 }



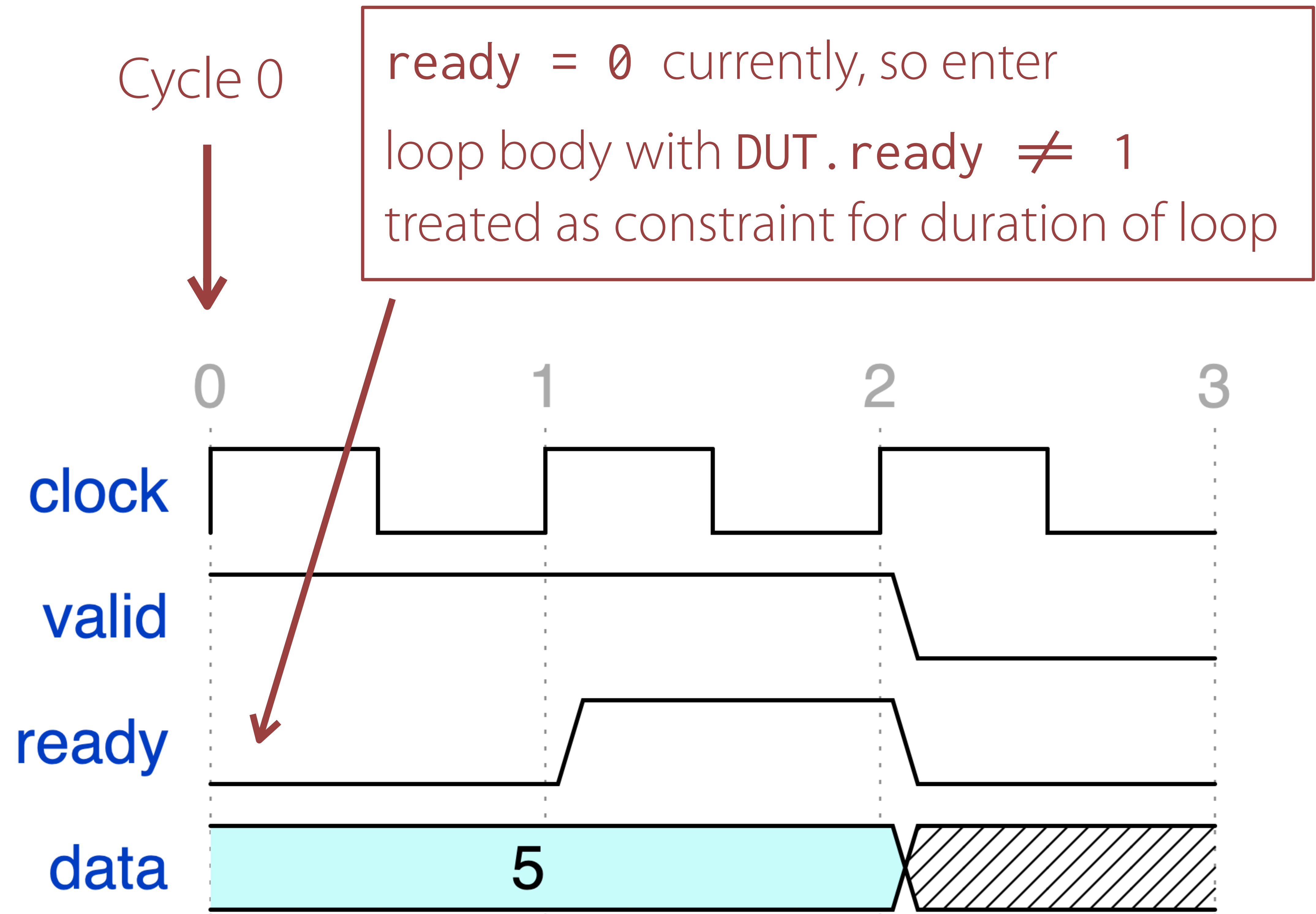
Constraints: { 1 = 1, data = 5 }

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



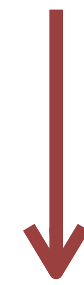
Constraints: { 1 = 1, data = 5 }

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



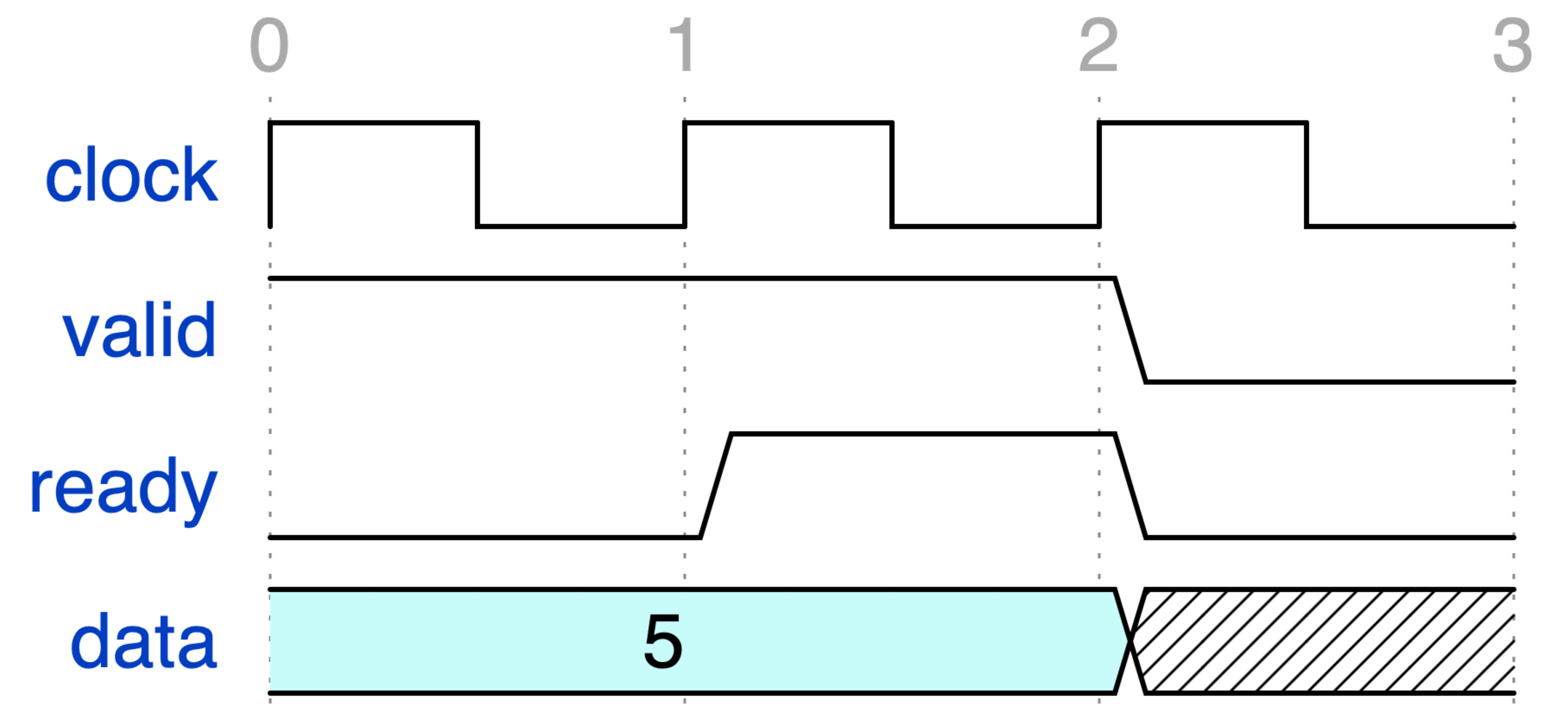
Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

Cycle 0

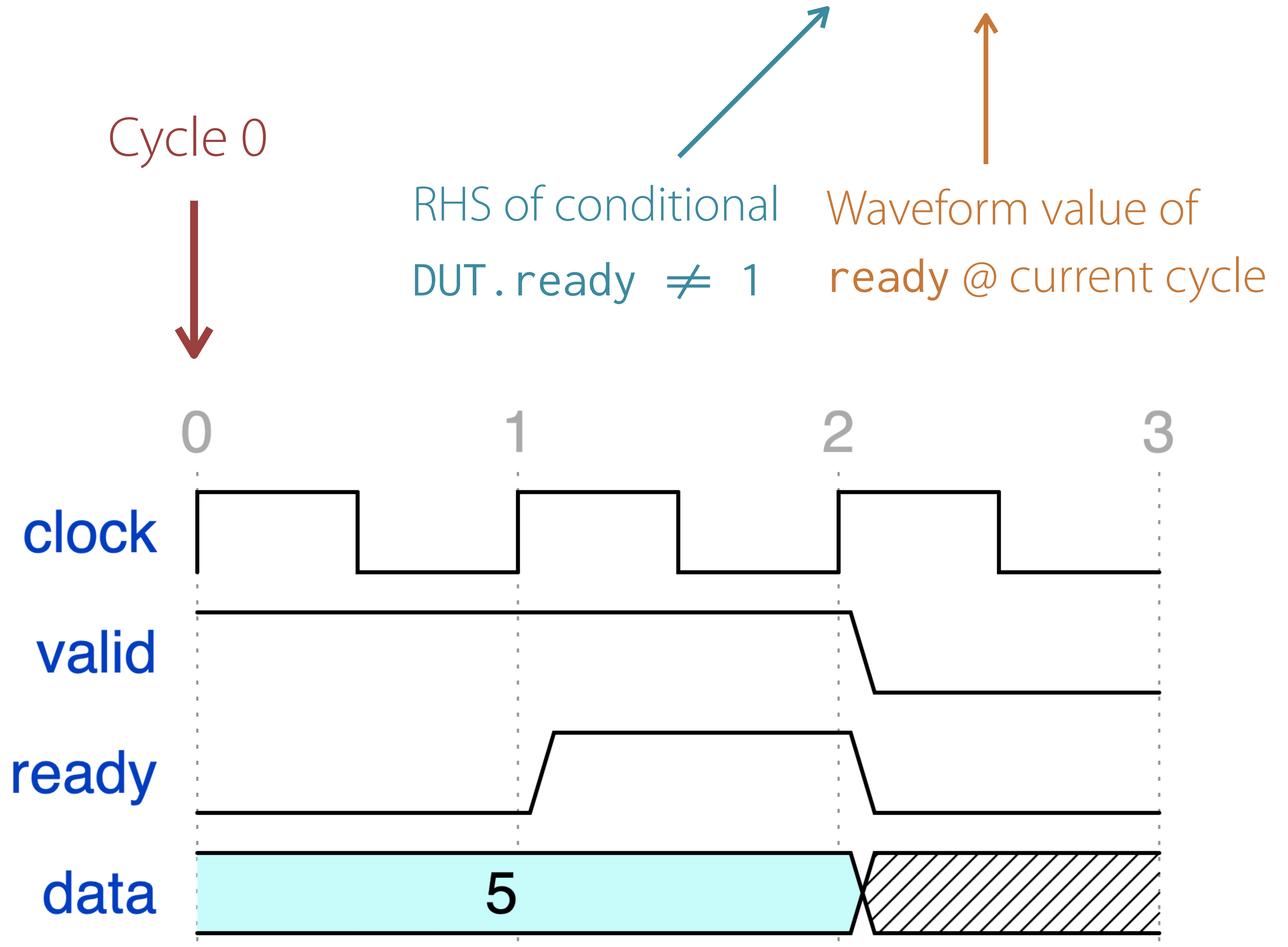


Add a new constraint 1 ≠ 0 representing the loop guard

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



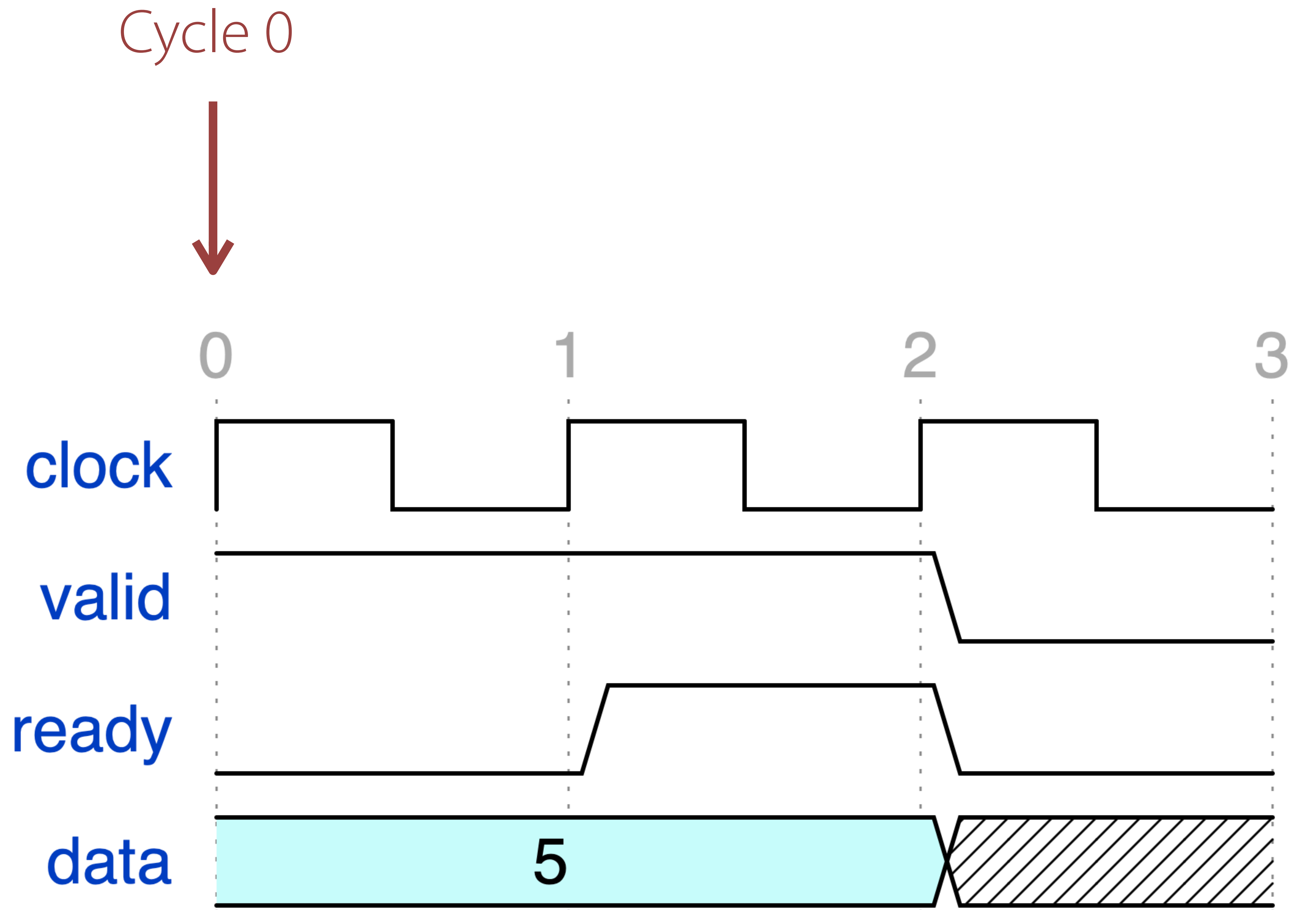
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```



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

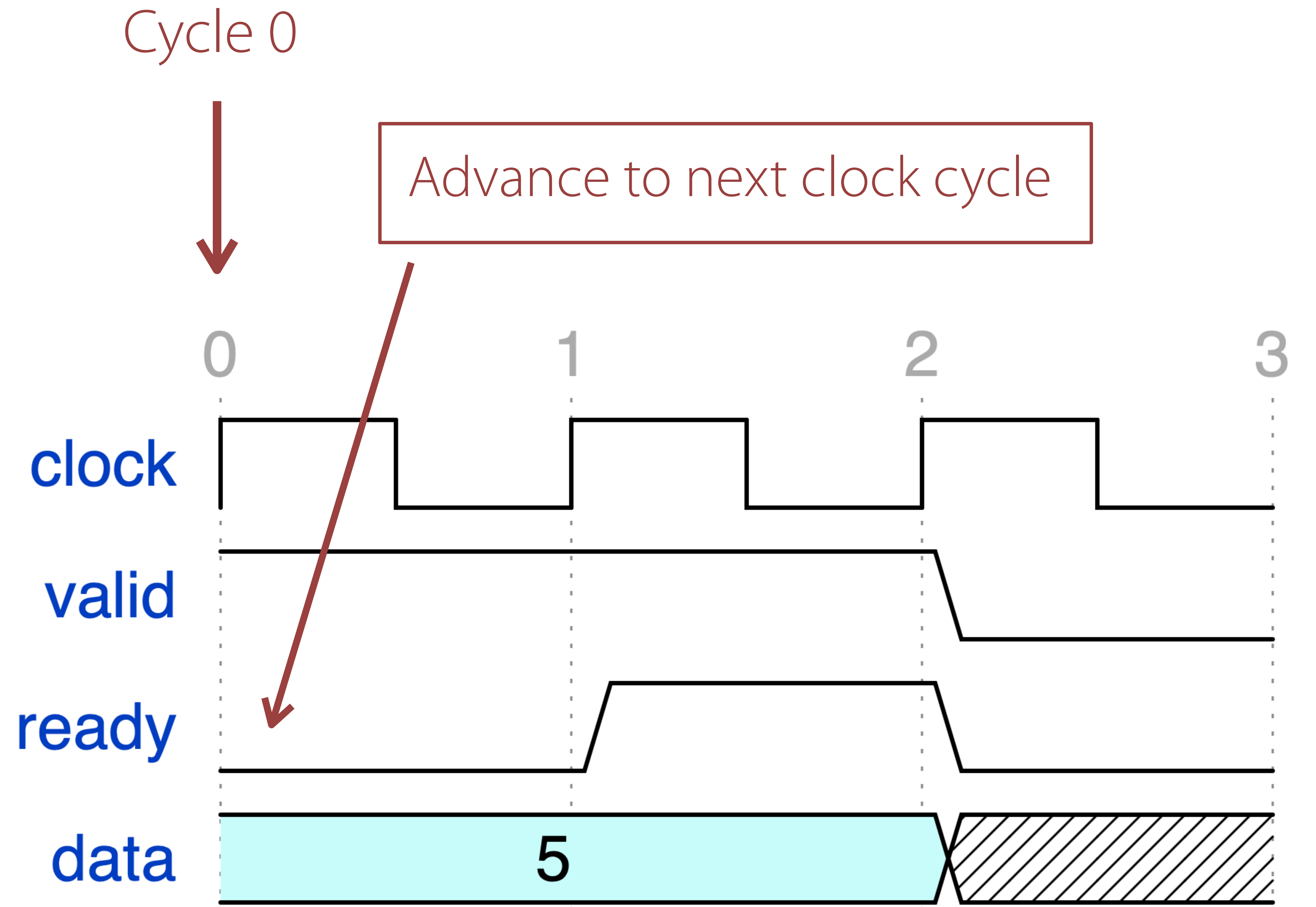
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```

Evaluate loop body



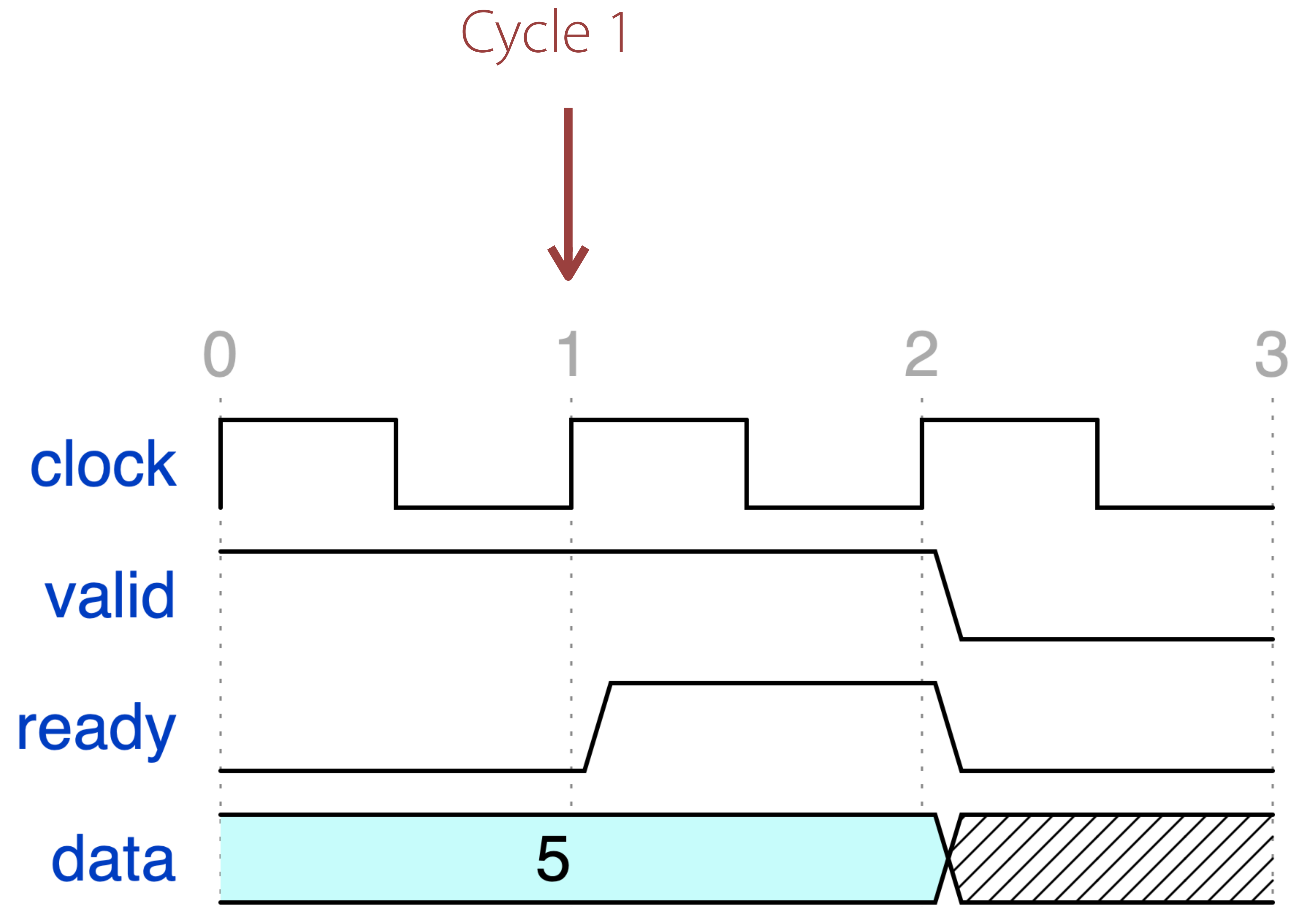
Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

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



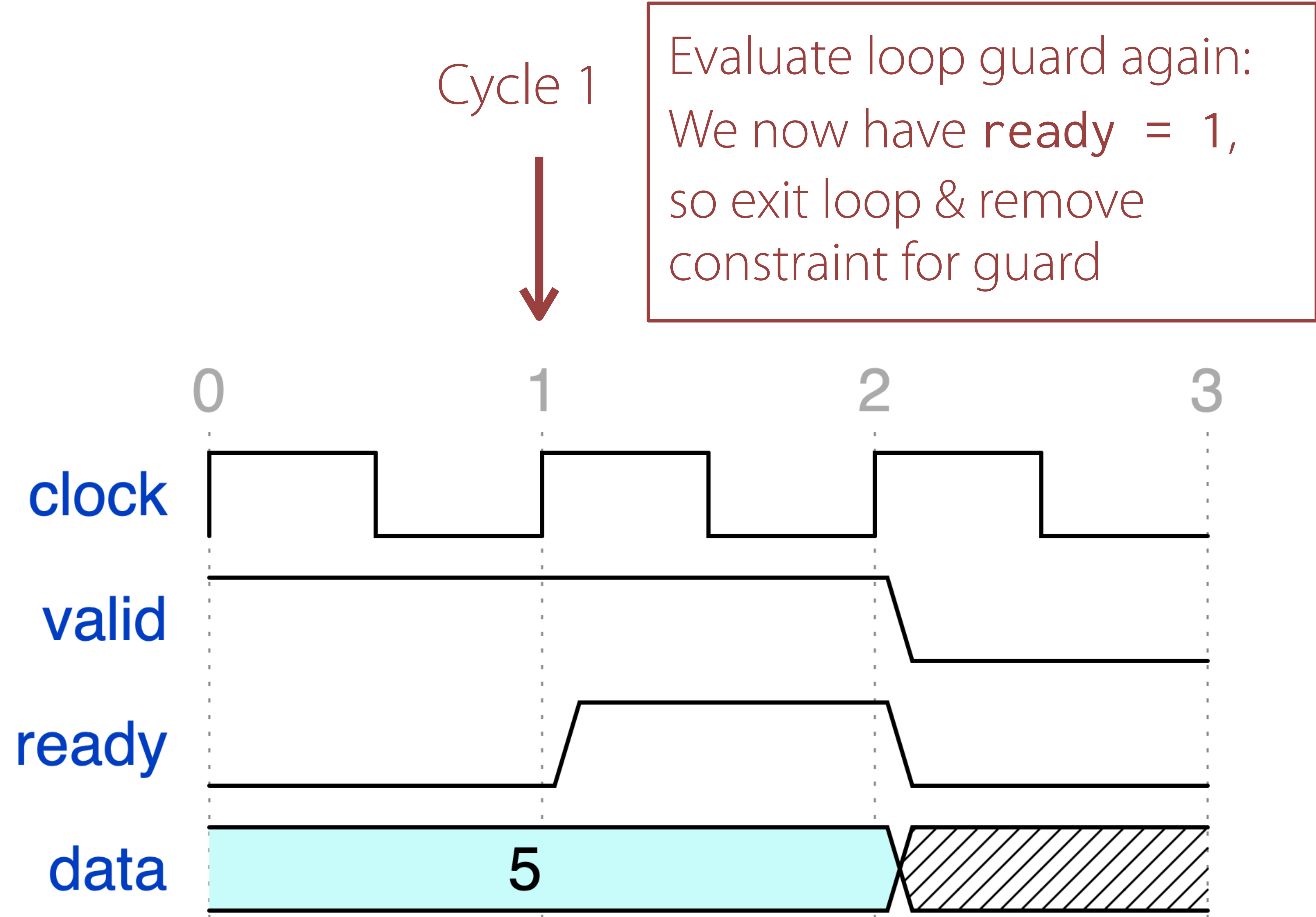
Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

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



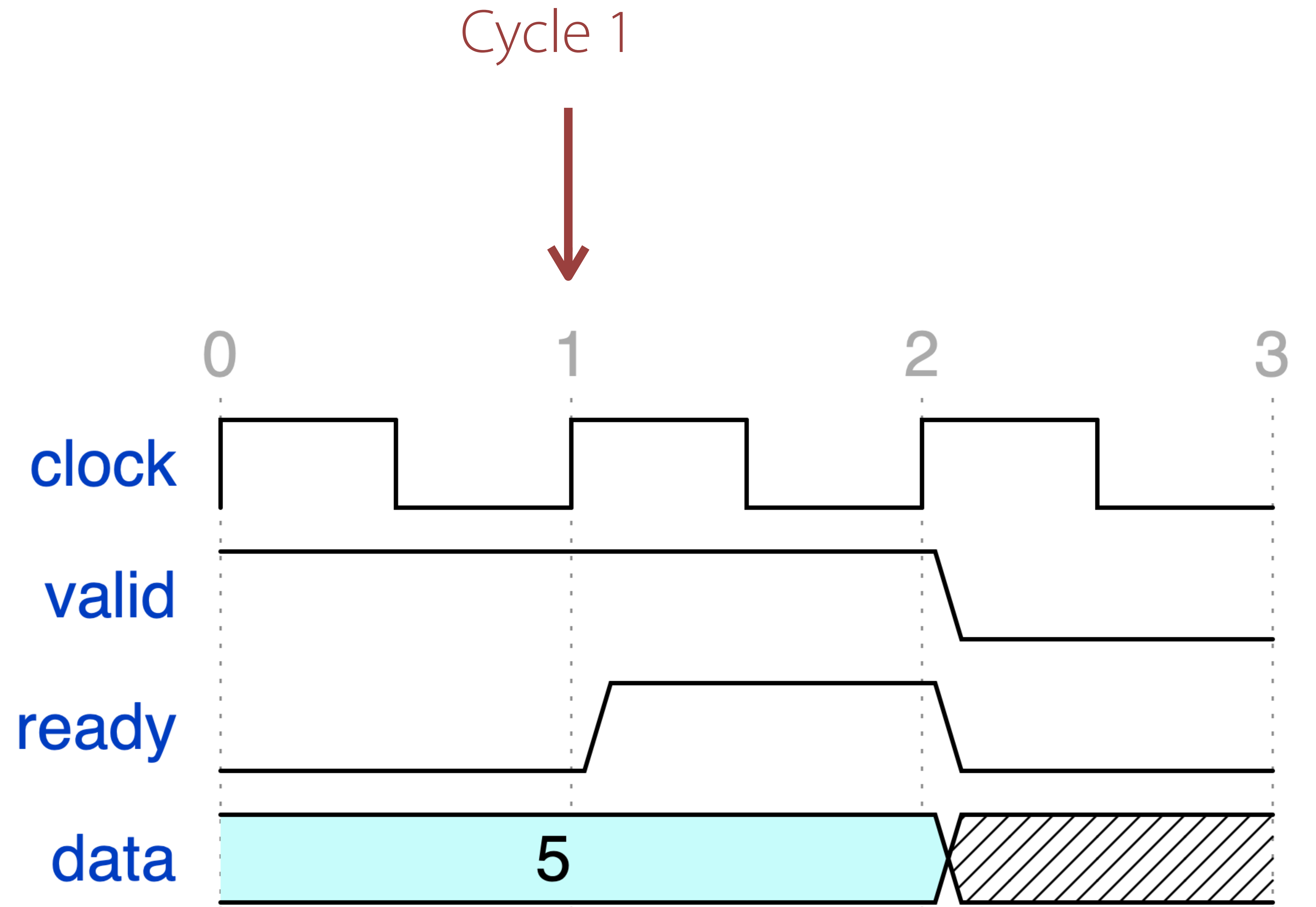
Constraints: { 1 = 1, data = 5, 1 ≠ 0 }

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

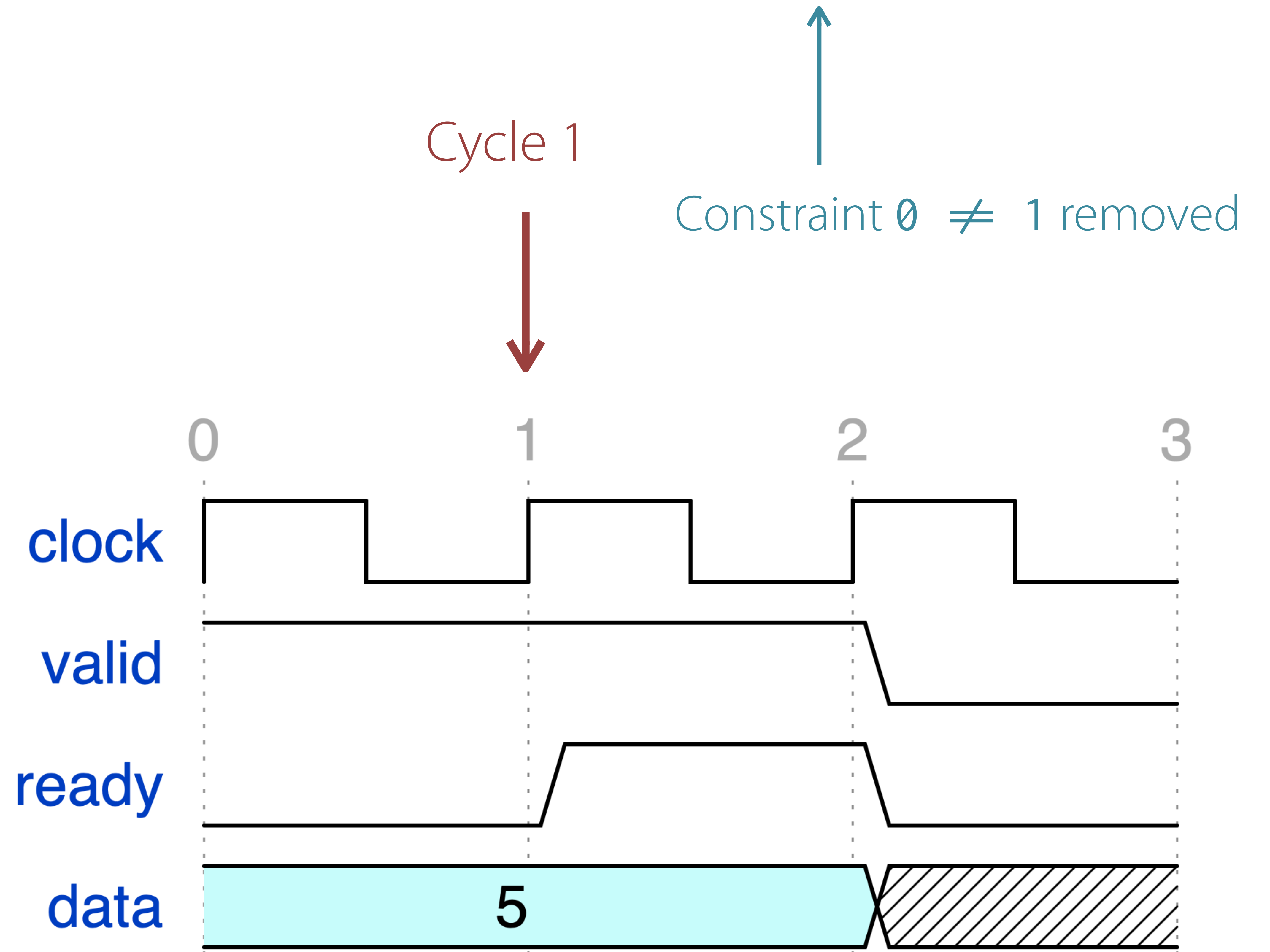


Constraints: { 1 = 1, data = 5 }

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



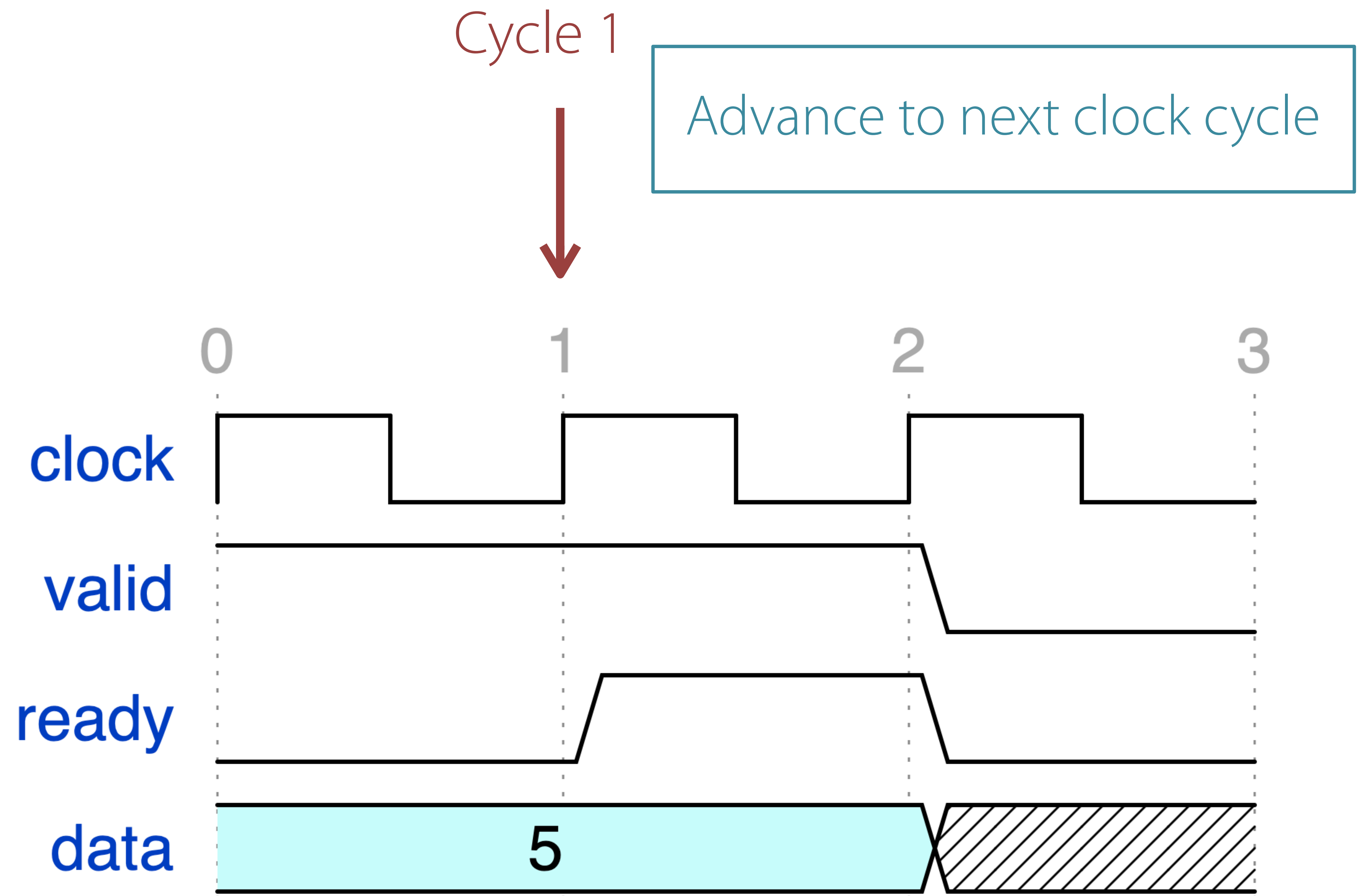
Constraints: { 1 = 1, data = 5 }




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

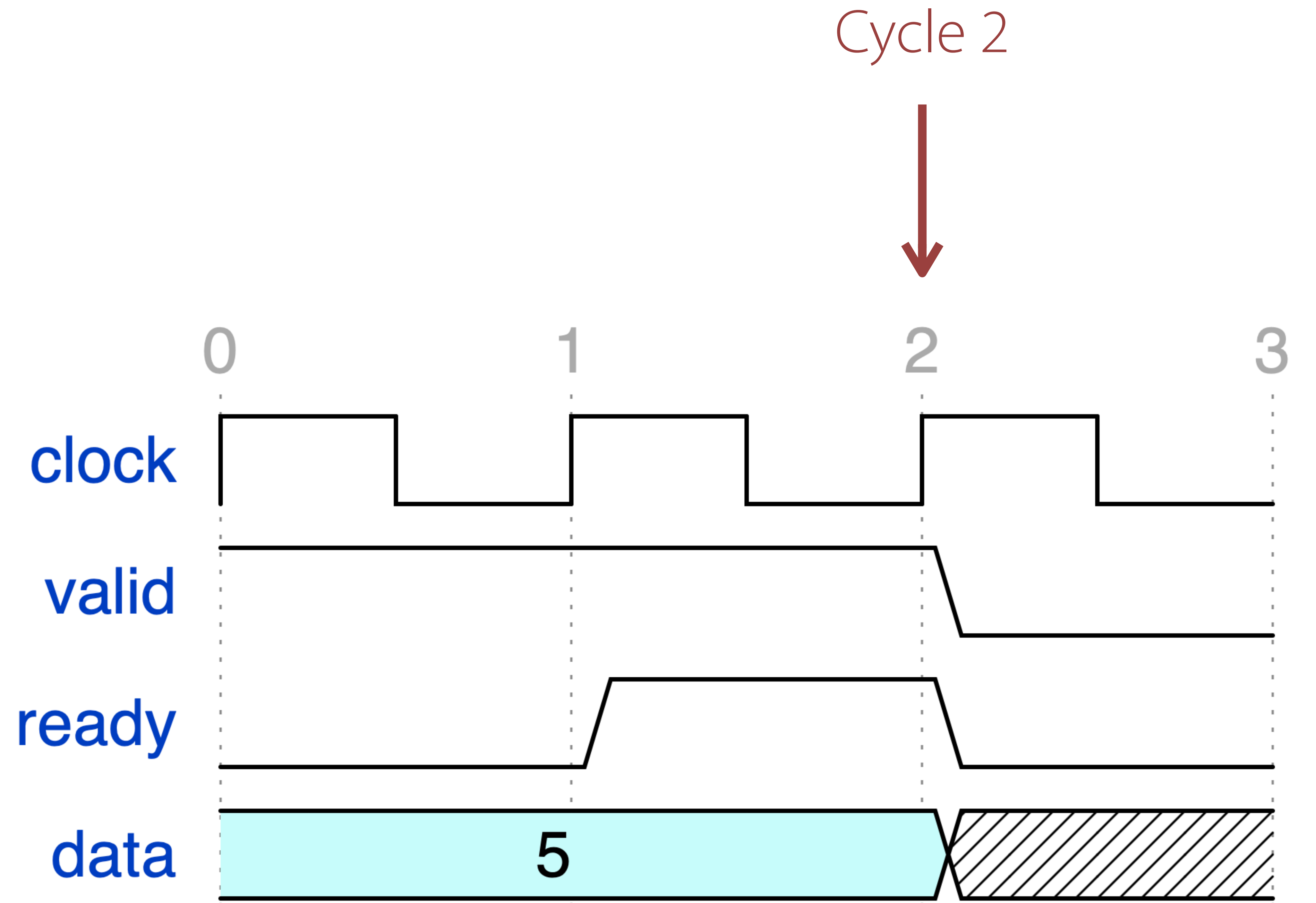
Constraints: { 1 = 1, data = 5 }

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



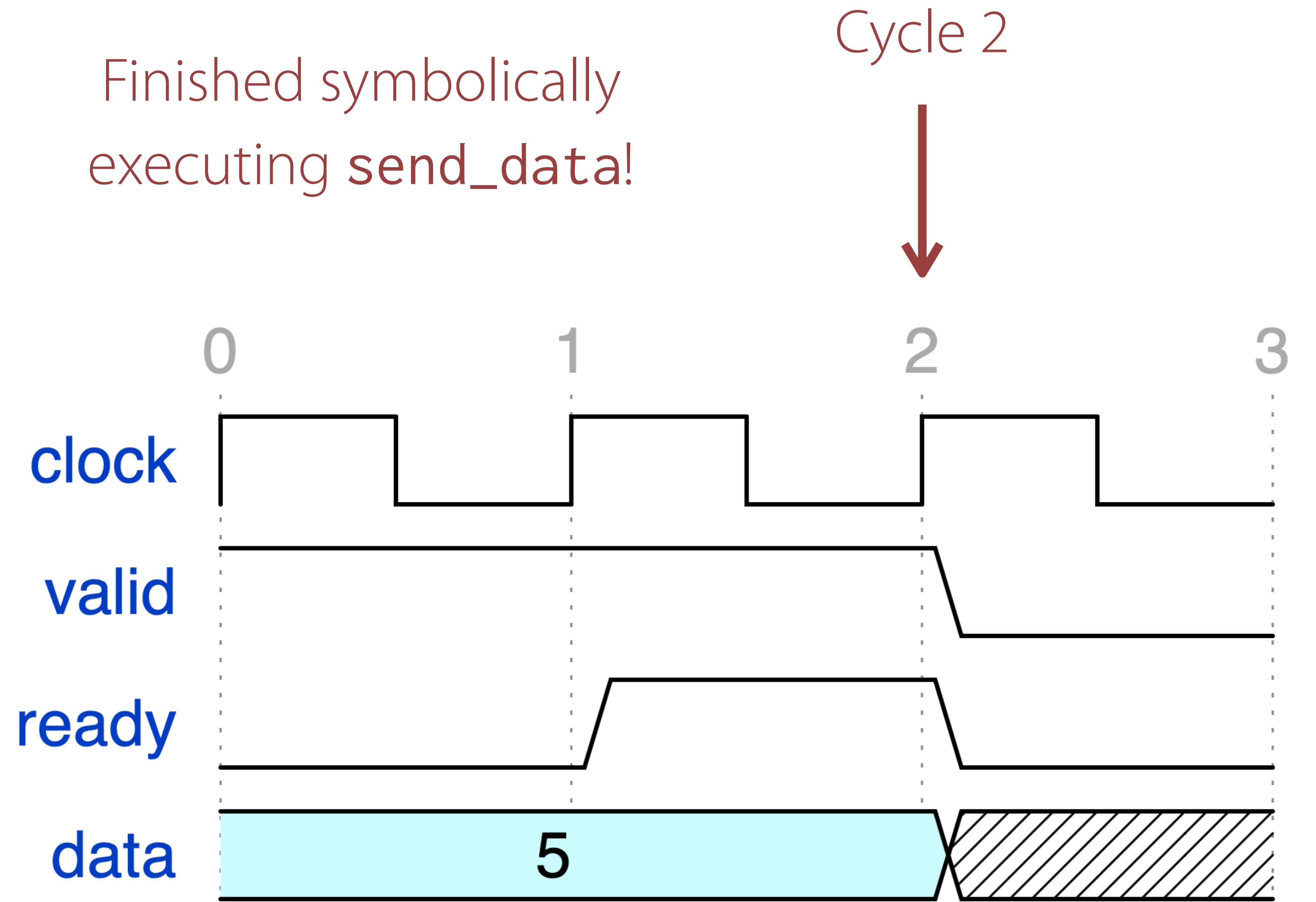
Constraints: { 1 = 1, data = 5 }

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



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

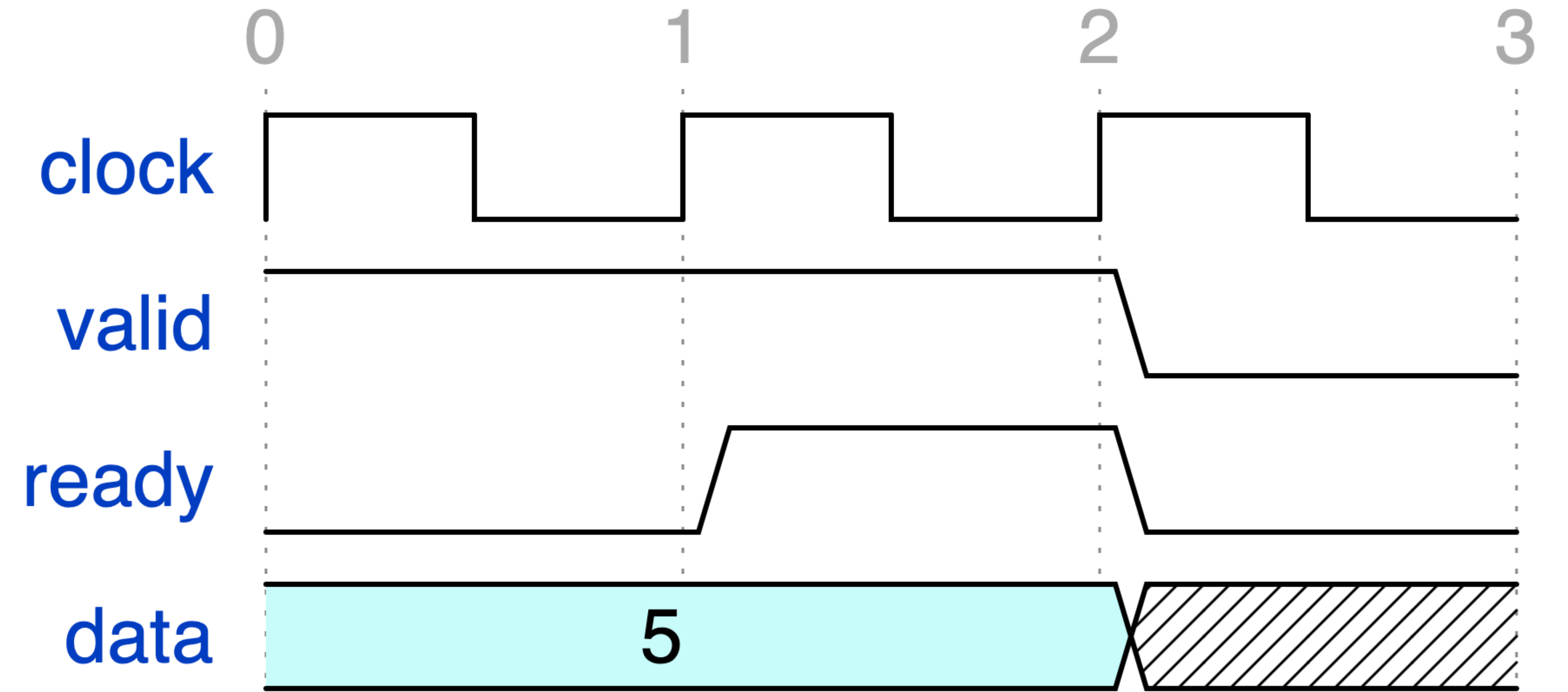
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
   DUT.valid := 1;
   DUT.data  := data;
   while (DUT.ready ≠ 1) {
     step();
   }
   step();
}
```



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

Inferred transaction trace:

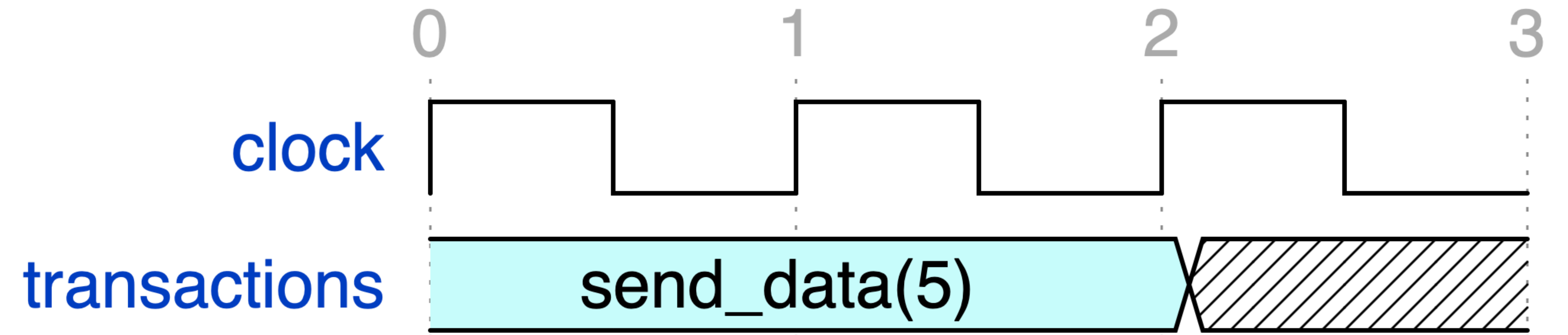
```
send_data(5); // cycles 0-2
```



Inferred transaction trace:

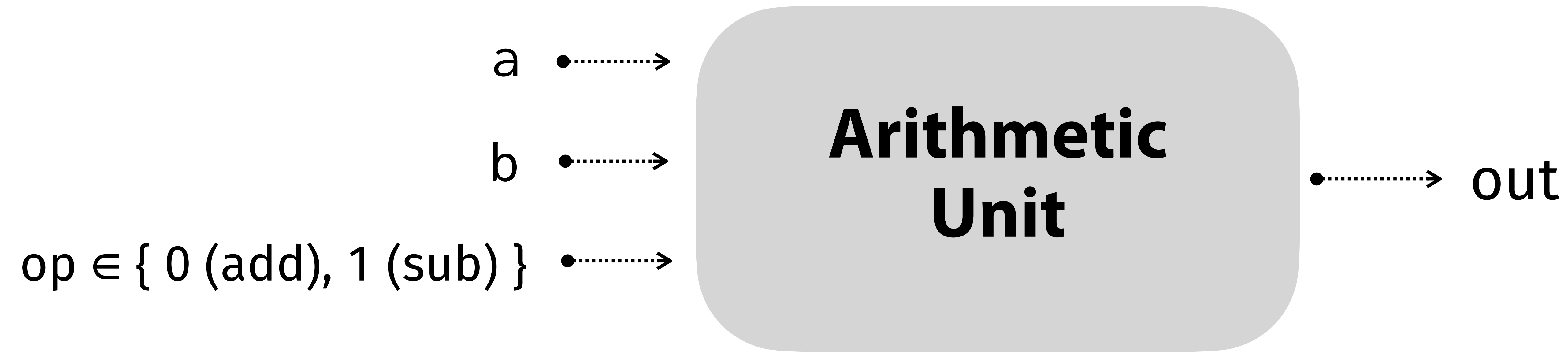
```
send_data(5); // cycles 0-2
```

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



A reconstructor example with multiple protocols

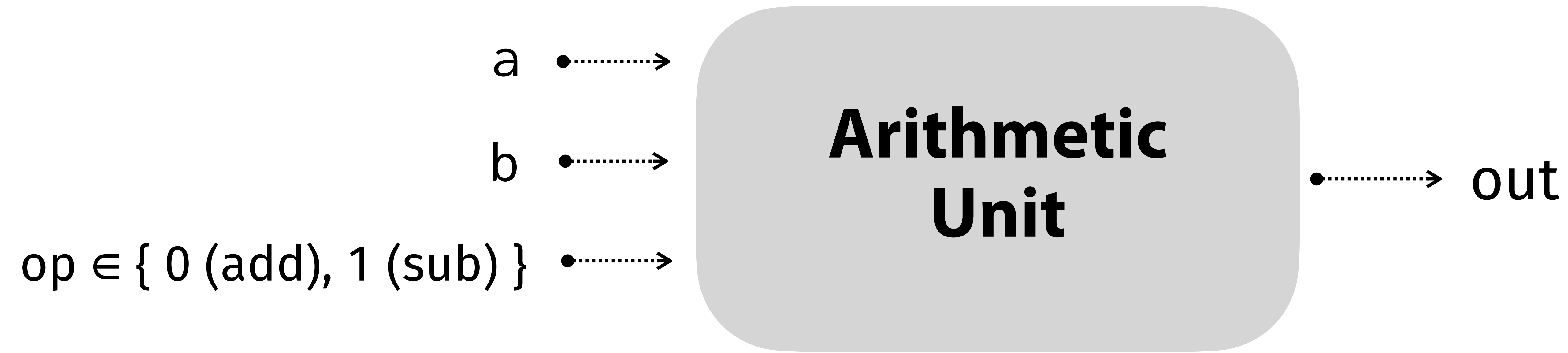
Example: Arithmetic Unit




```
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



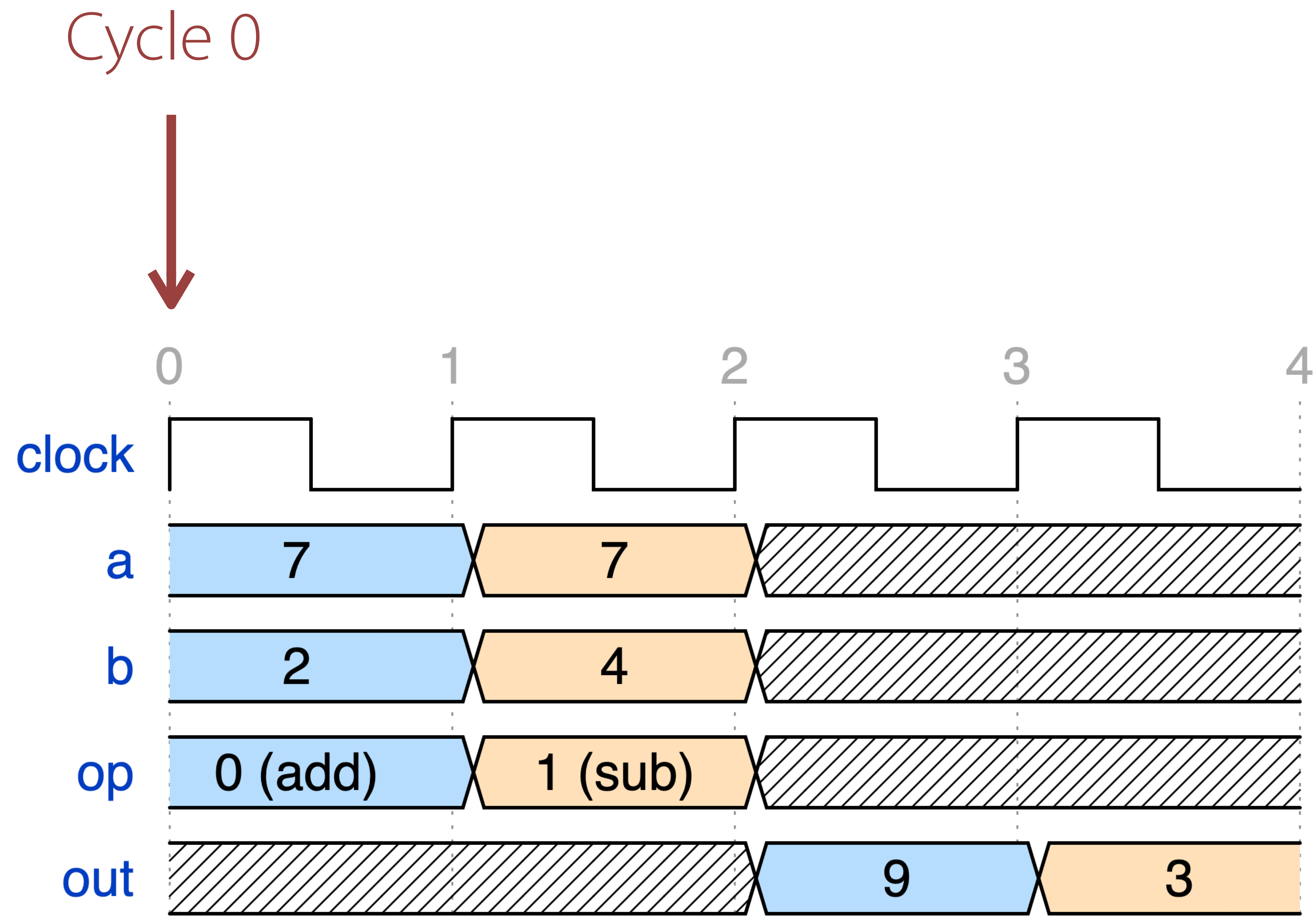

```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a; DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

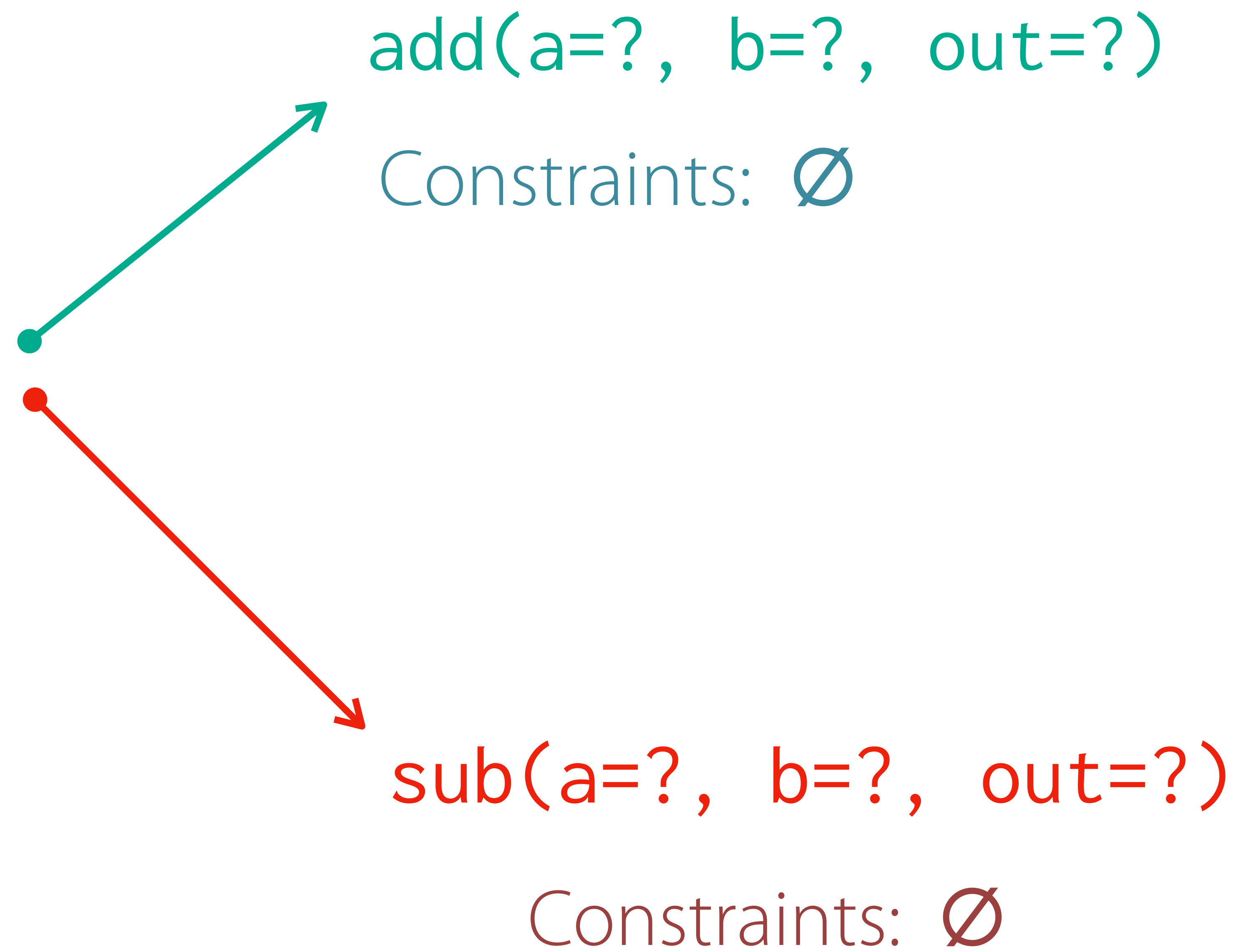
```
prot sub<DUT: ALU>(a, b, out) {
  DUT.a := a; DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

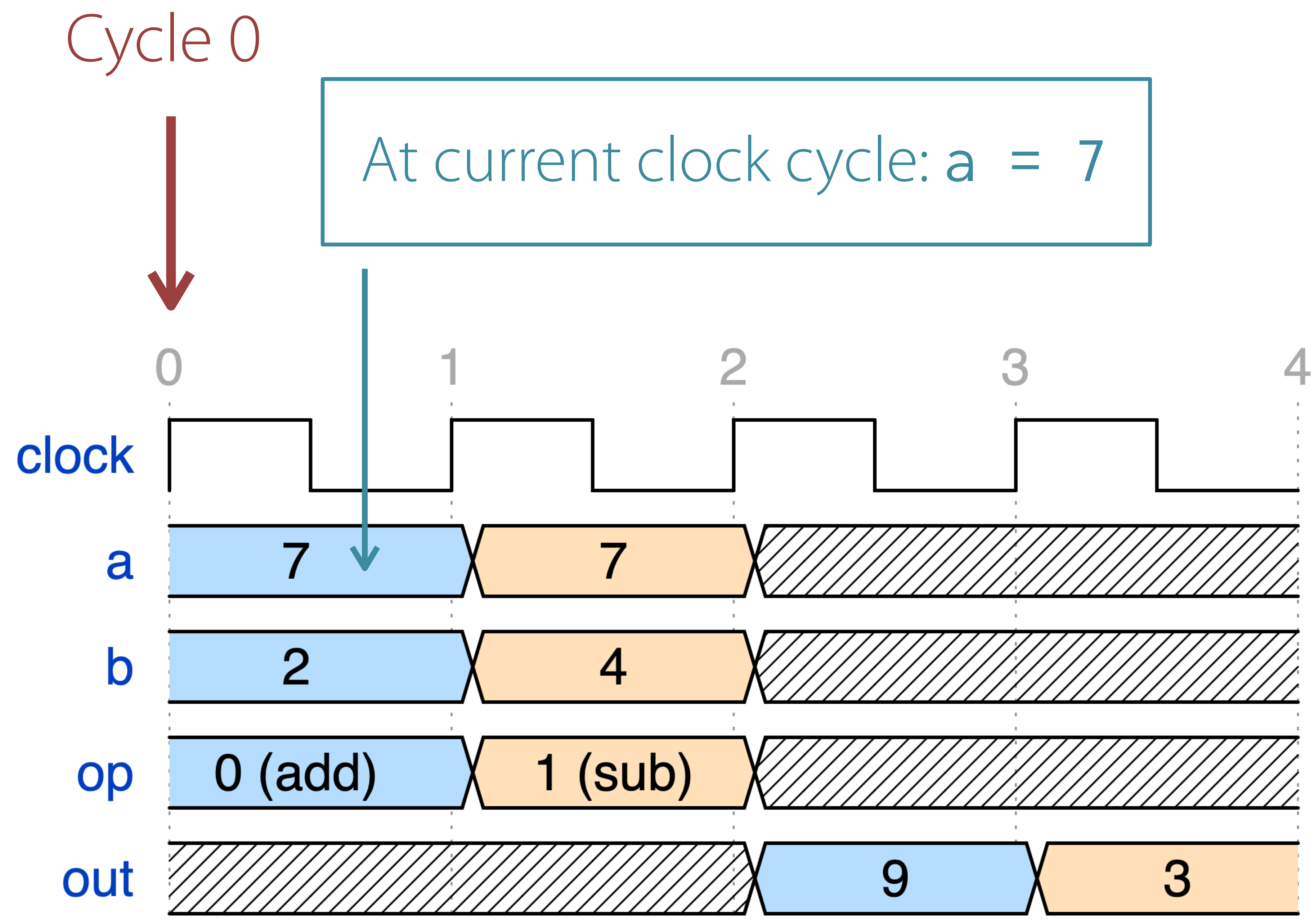
```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

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

Cycle 0

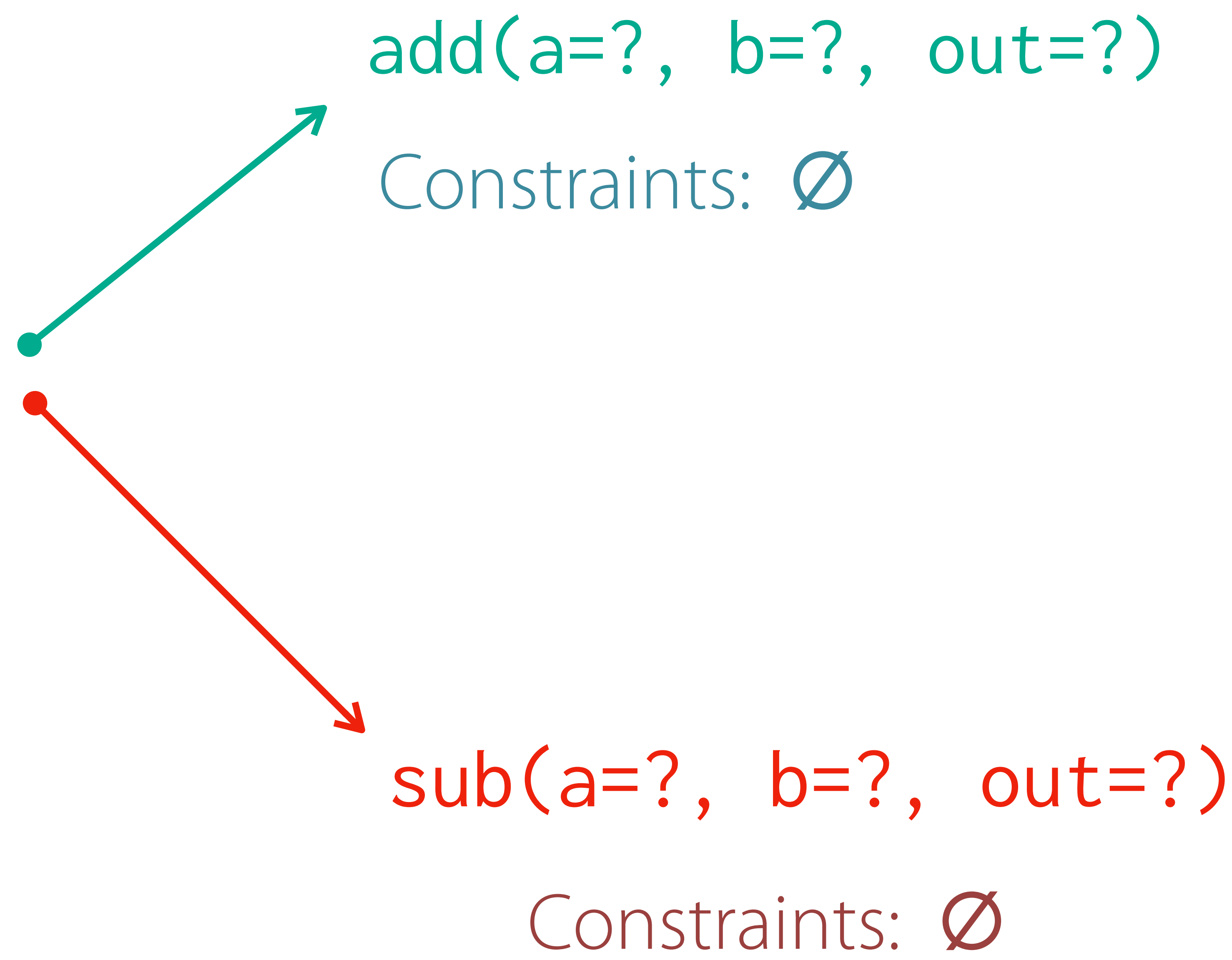






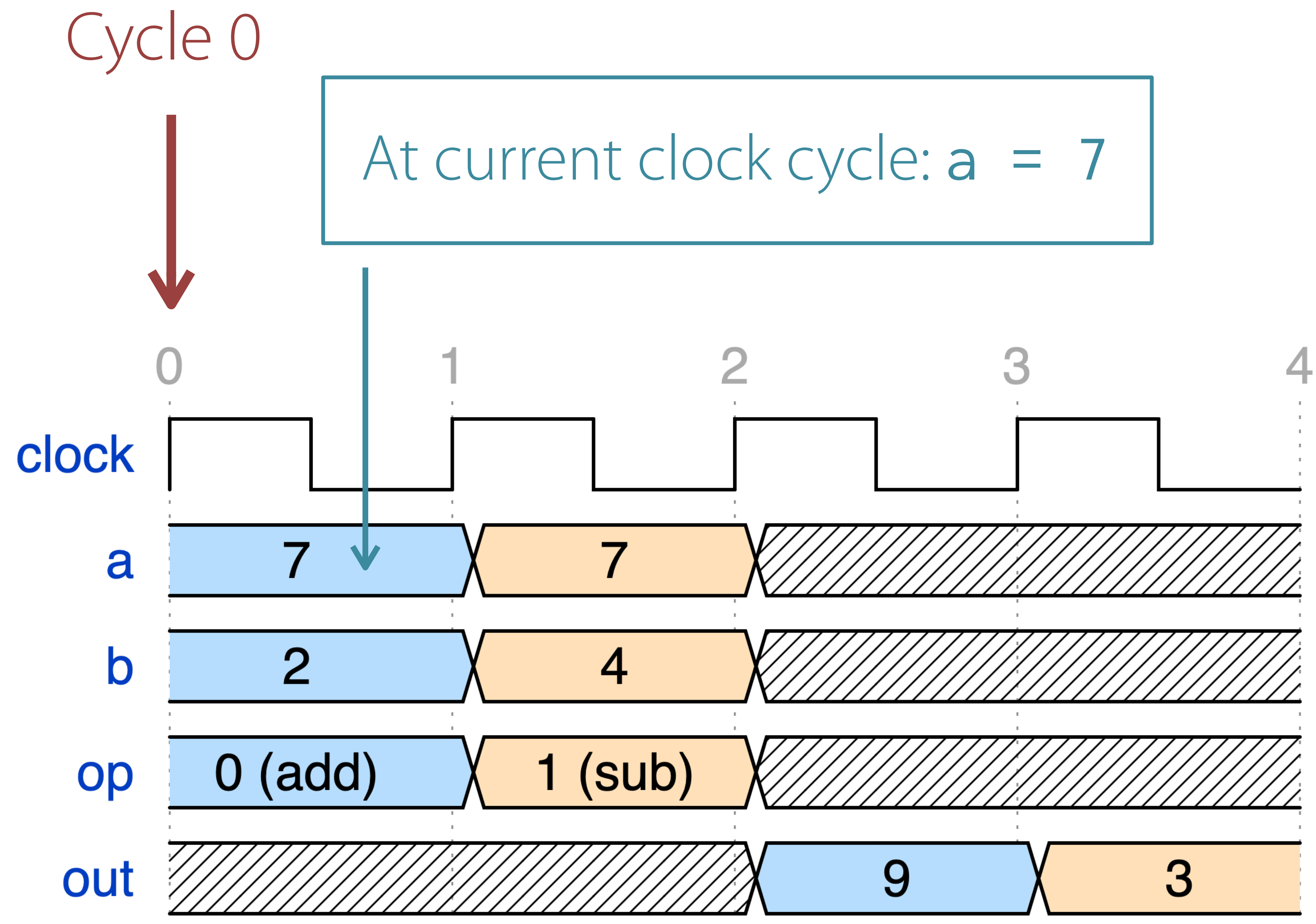

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

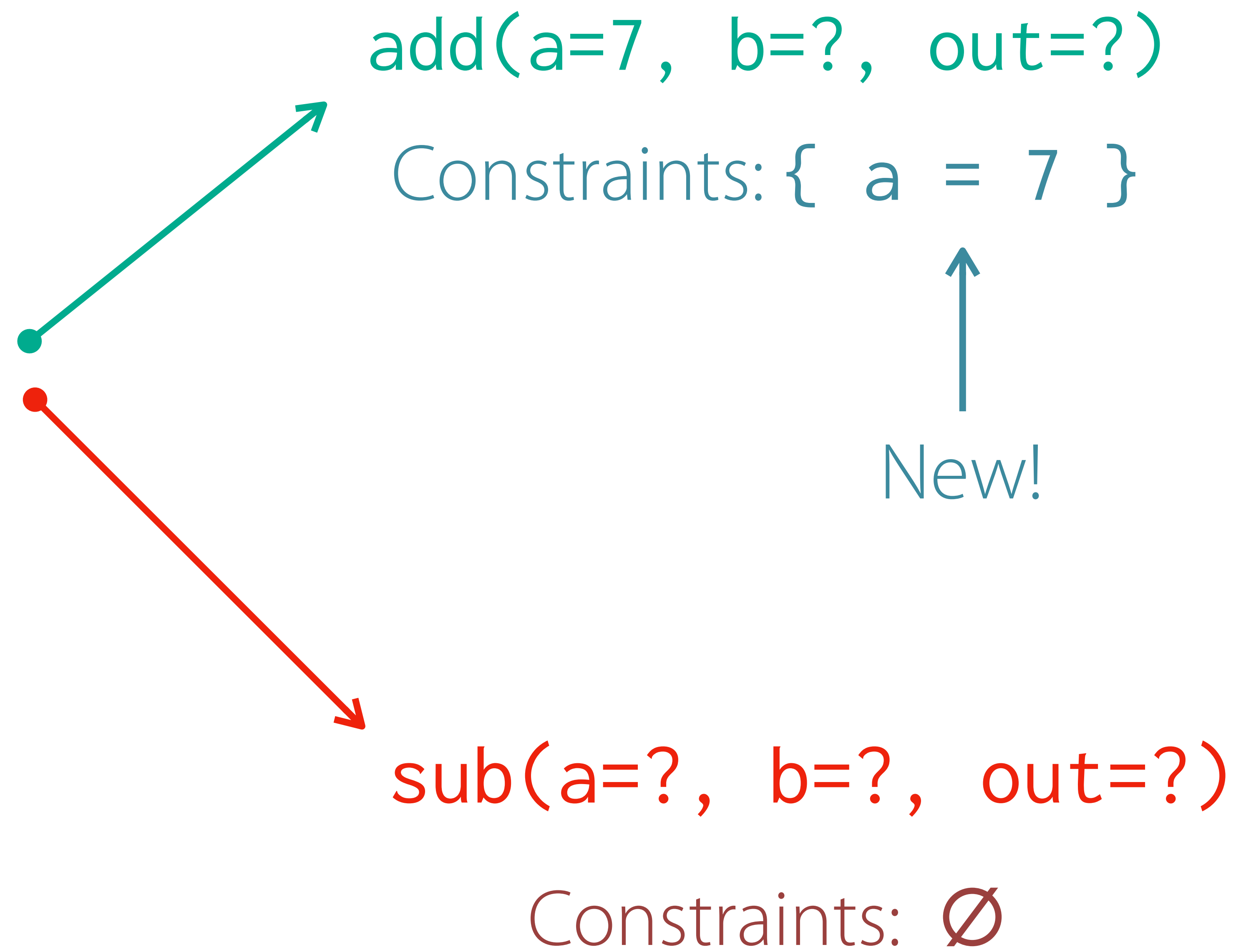
```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```



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

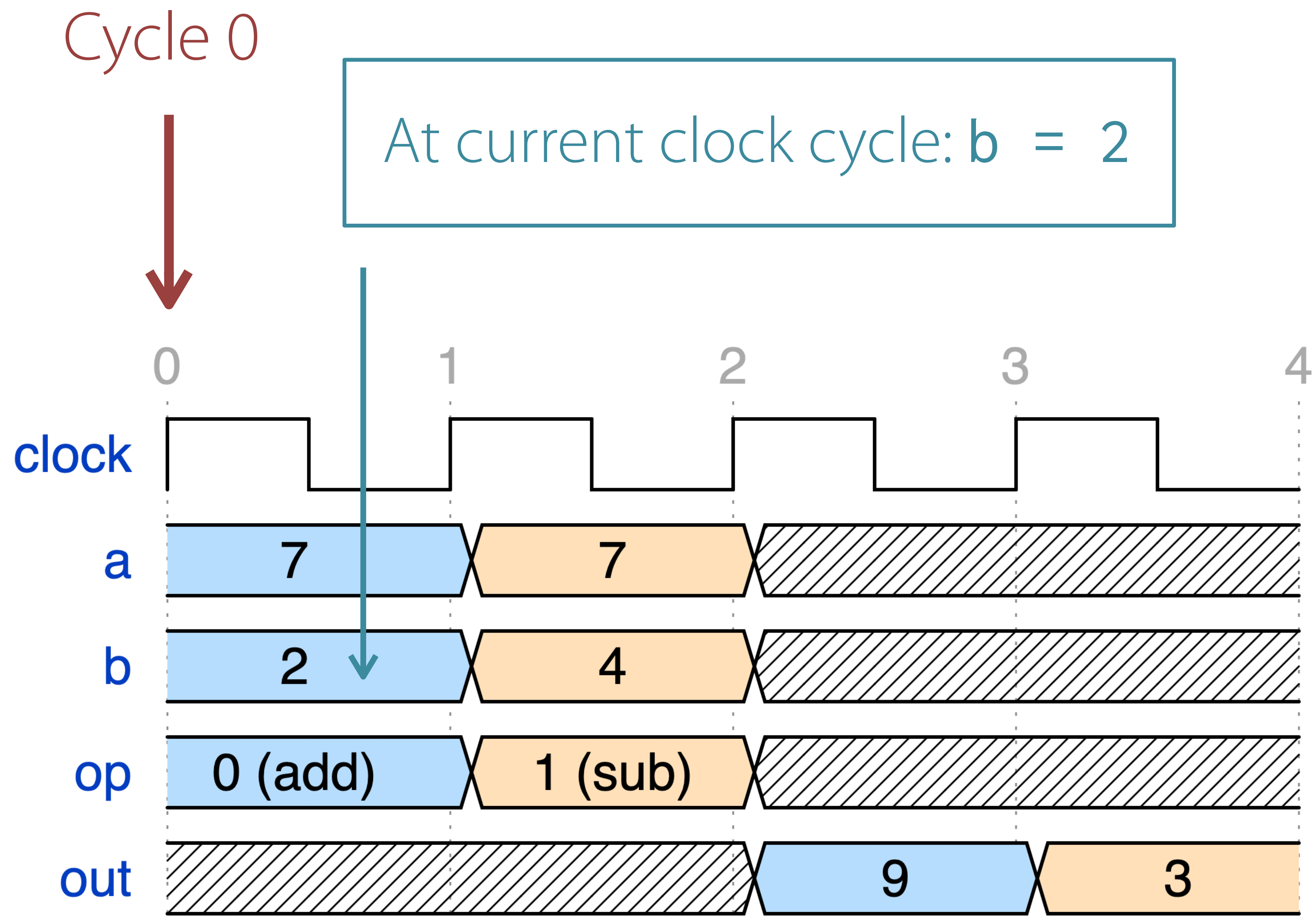
```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```





```
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=?, out=?)

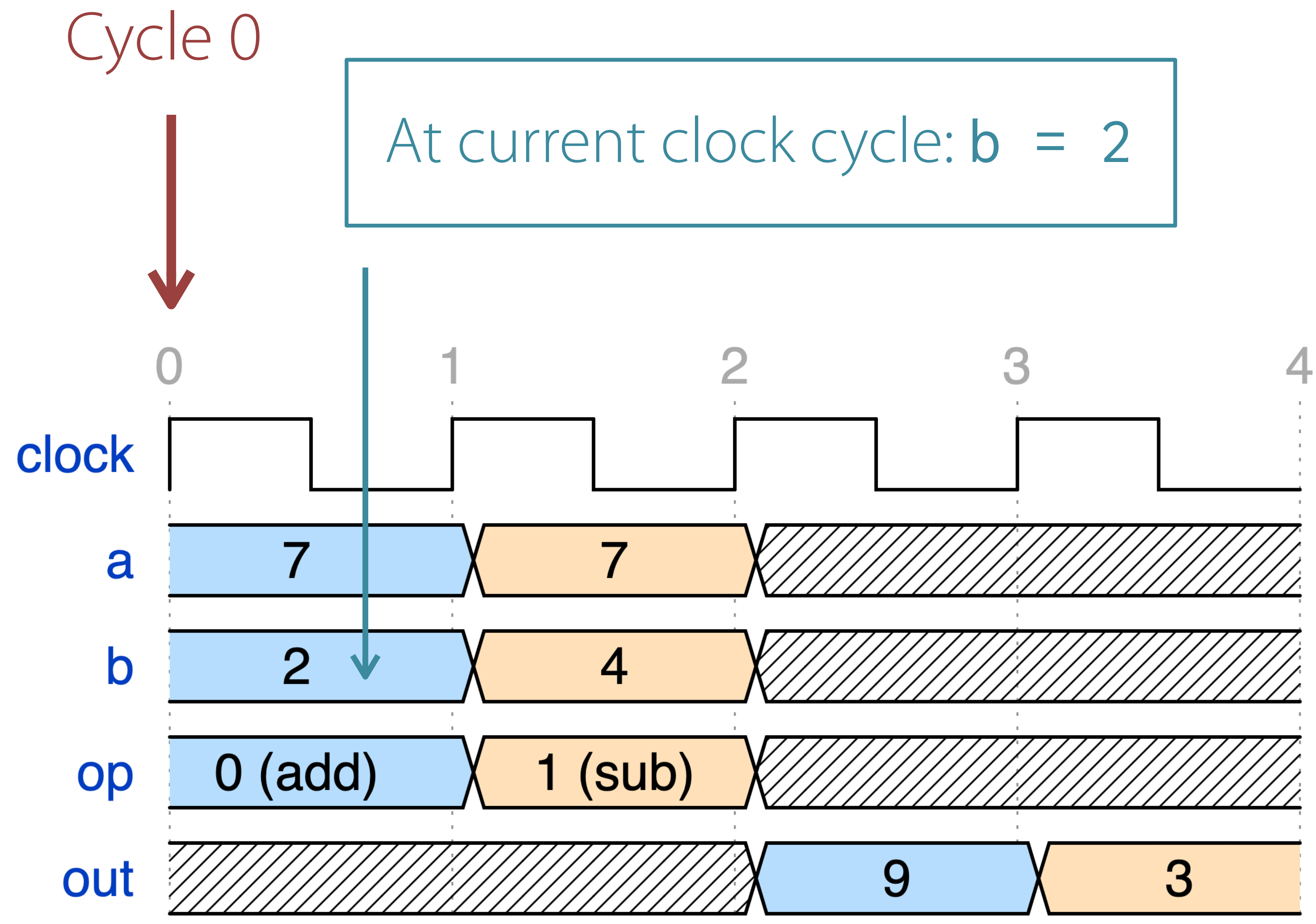
Constraints: { a = 7 }

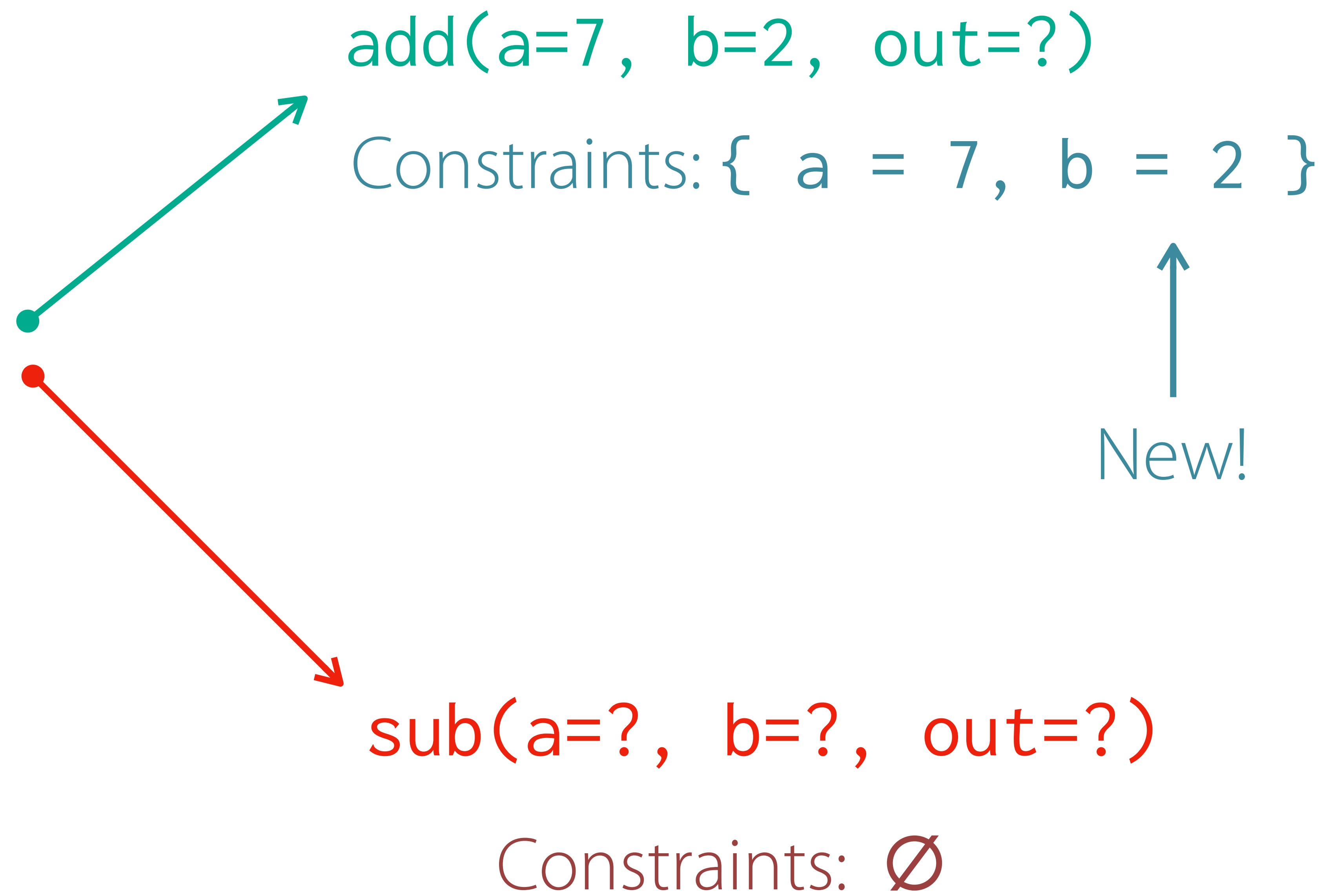
sub(a=?, b=?, out=?)

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();
  …
```



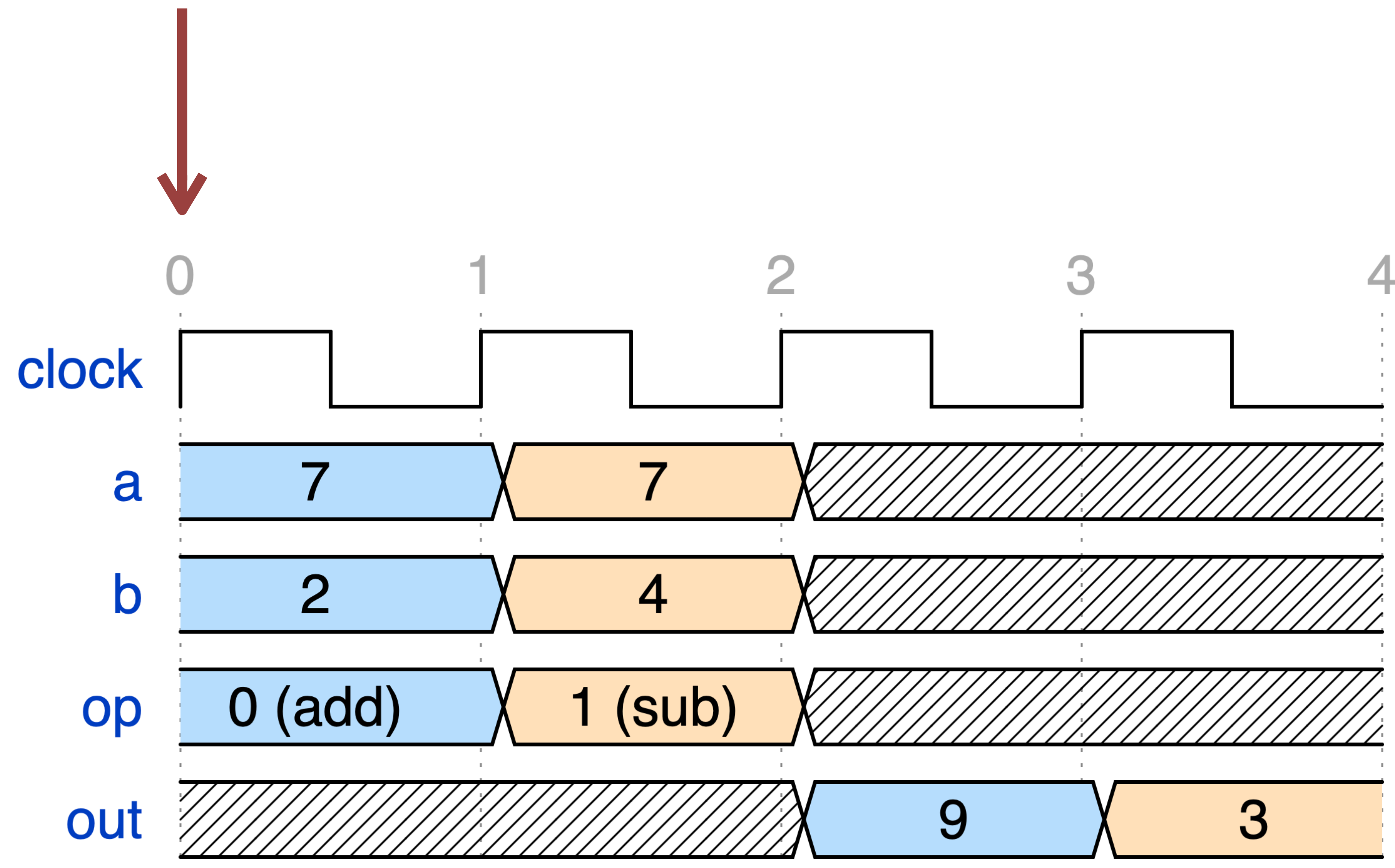


Check if current waveform value of `op =? 0`

```
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: ∅

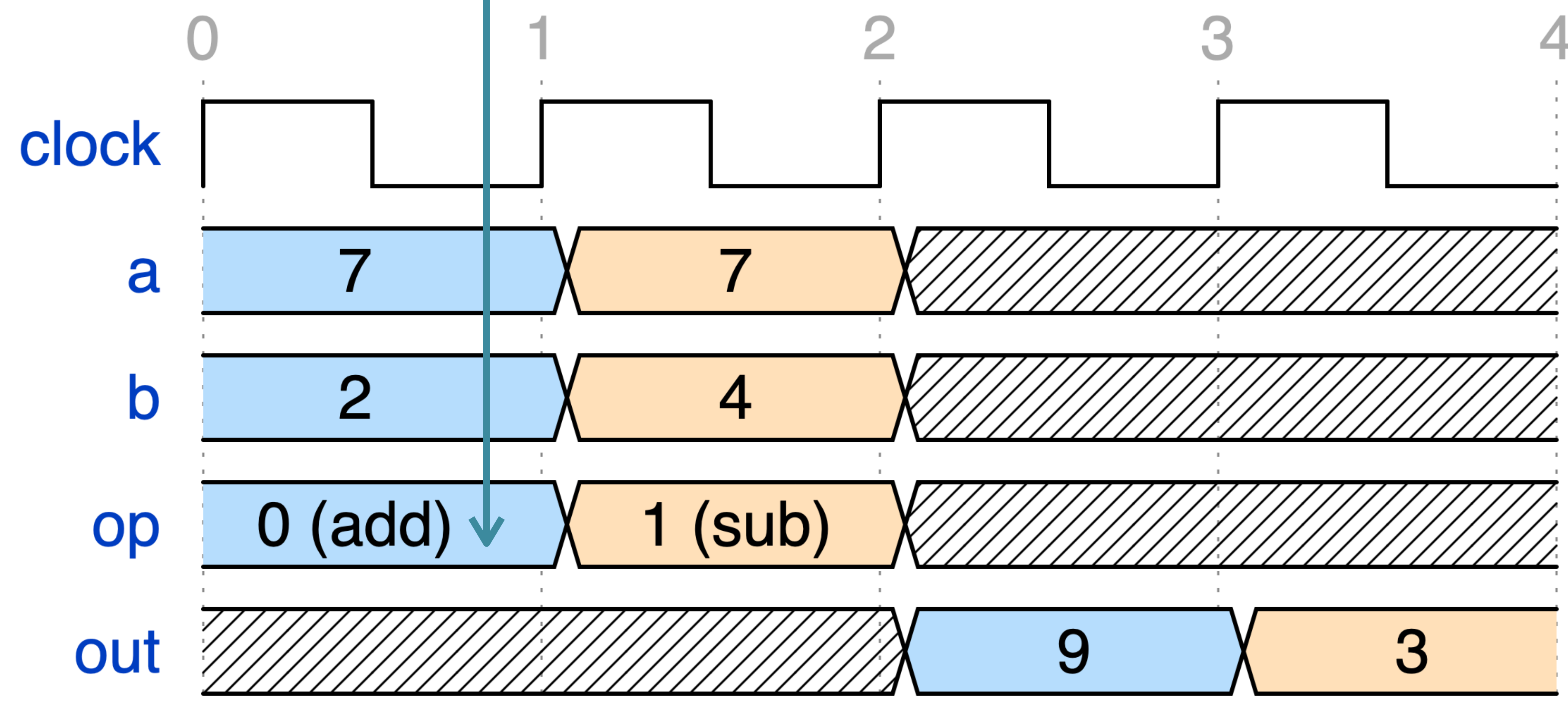
Check if current waveform value of `op =? 0`

```
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: ∅

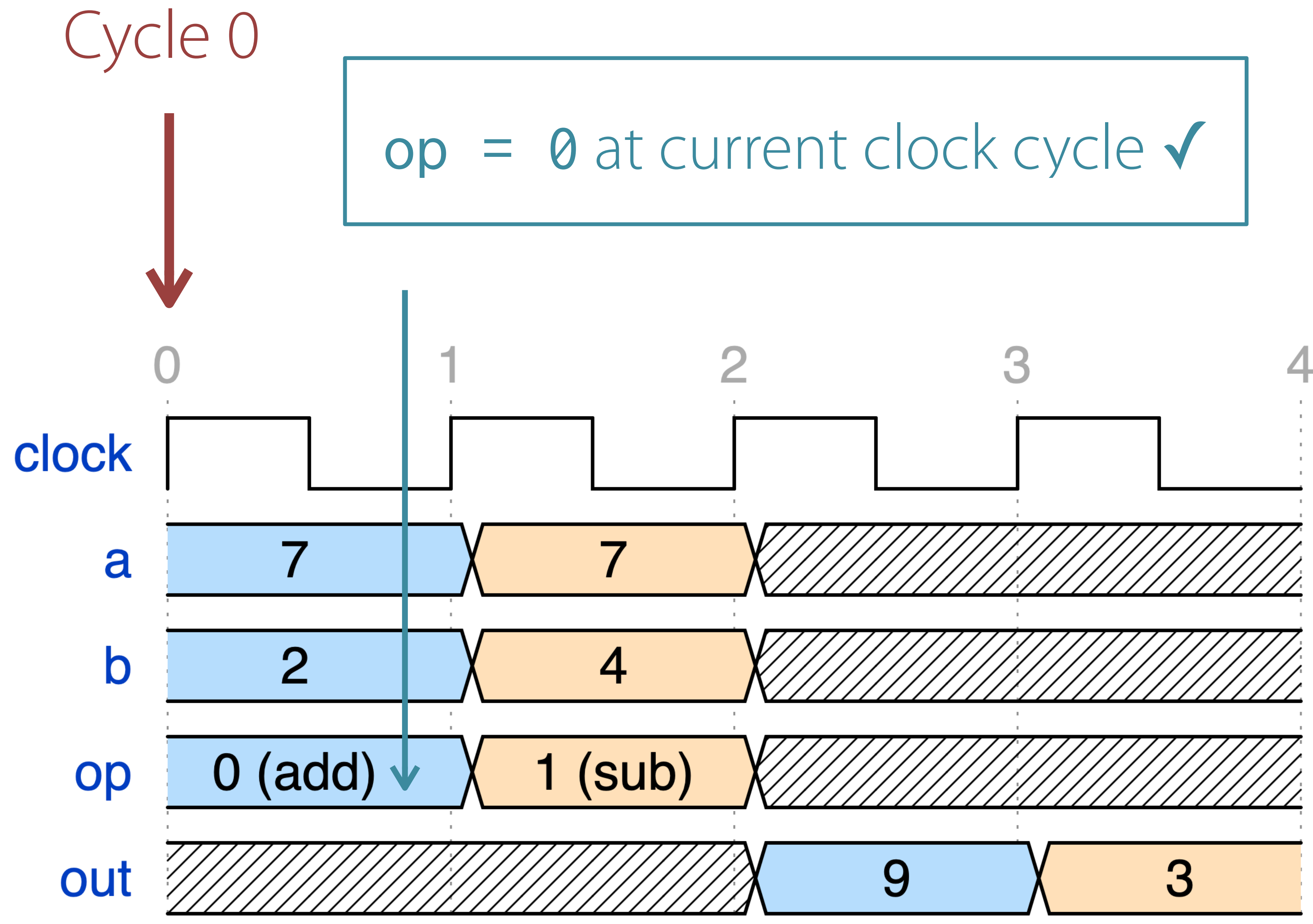
Check if current waveform value of `op =? 0`

```
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: ∅

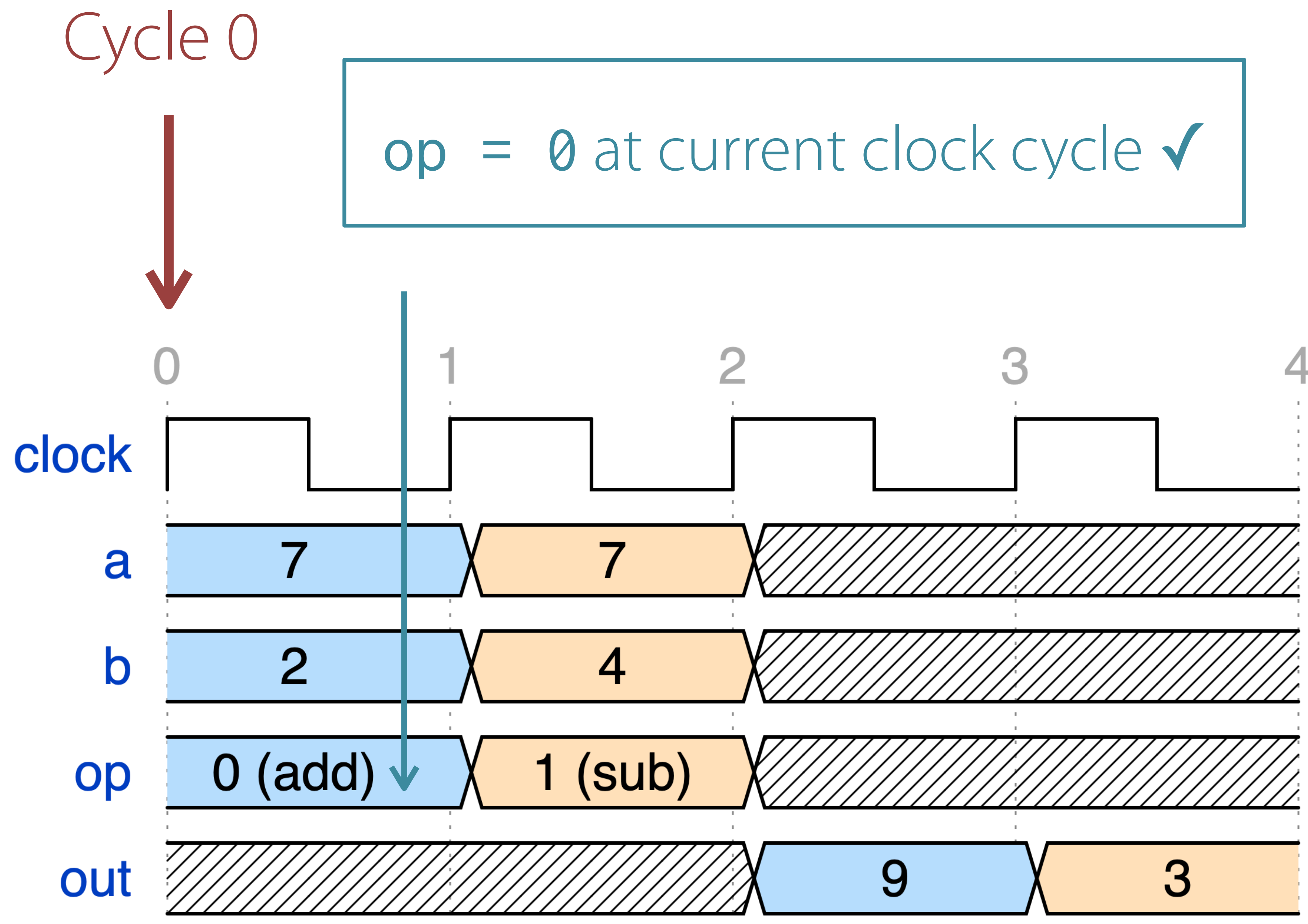
Check if current waveform value of `op =? 0`

```
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

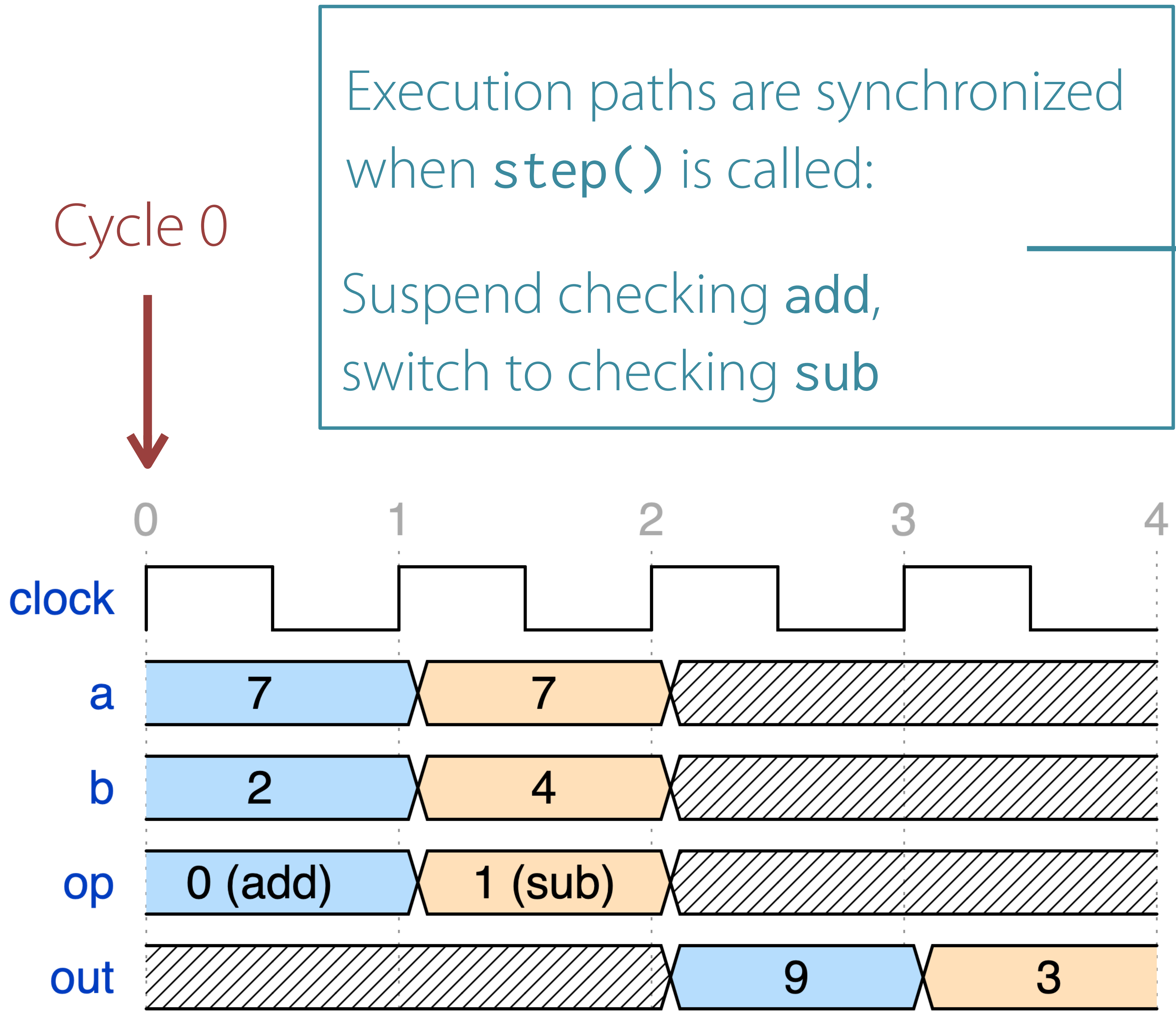
add(a=7, b=2, out=?)

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

RHS of DUT.op := 0

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

Constraints: ∅




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

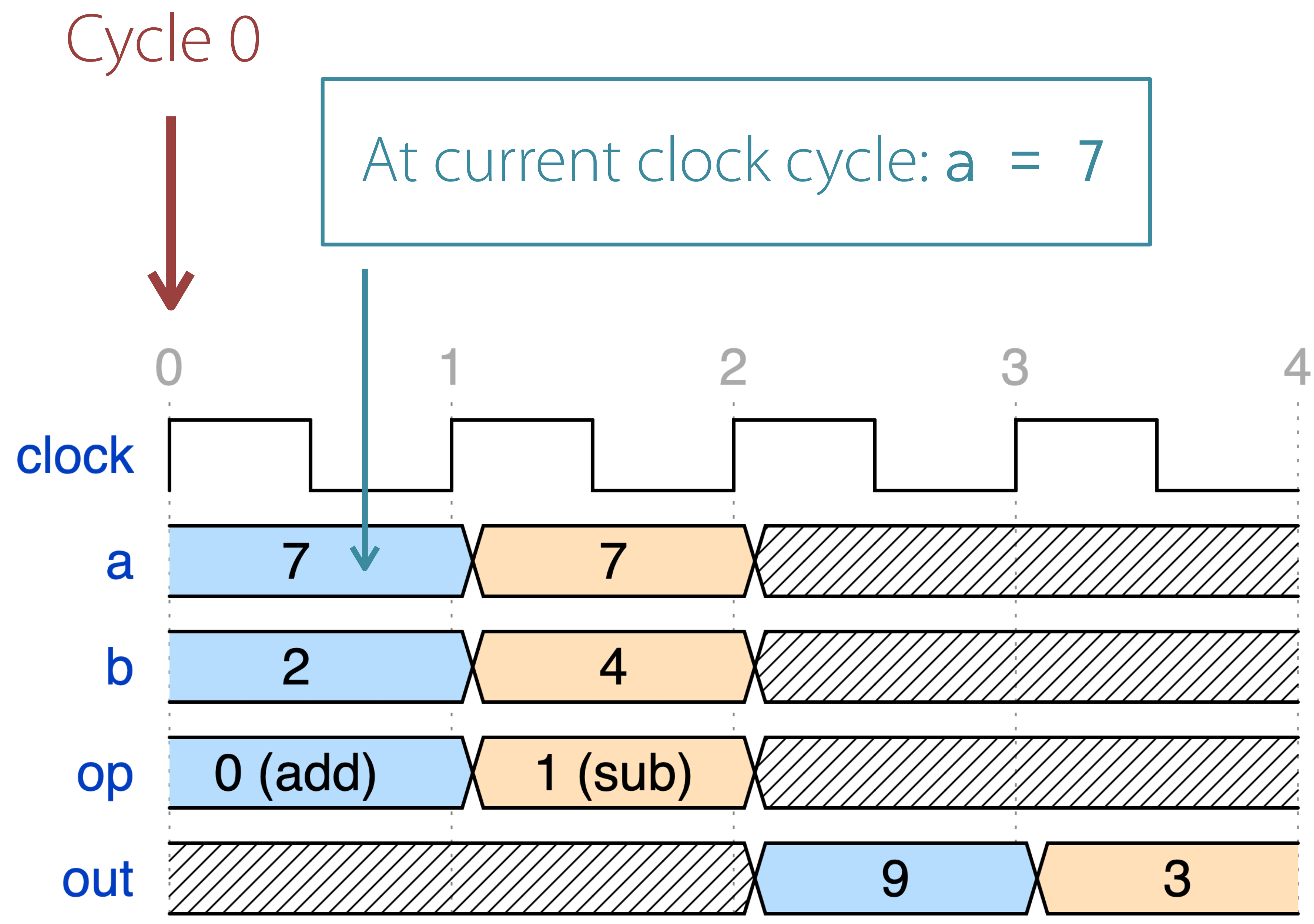
```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

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();
  …
```

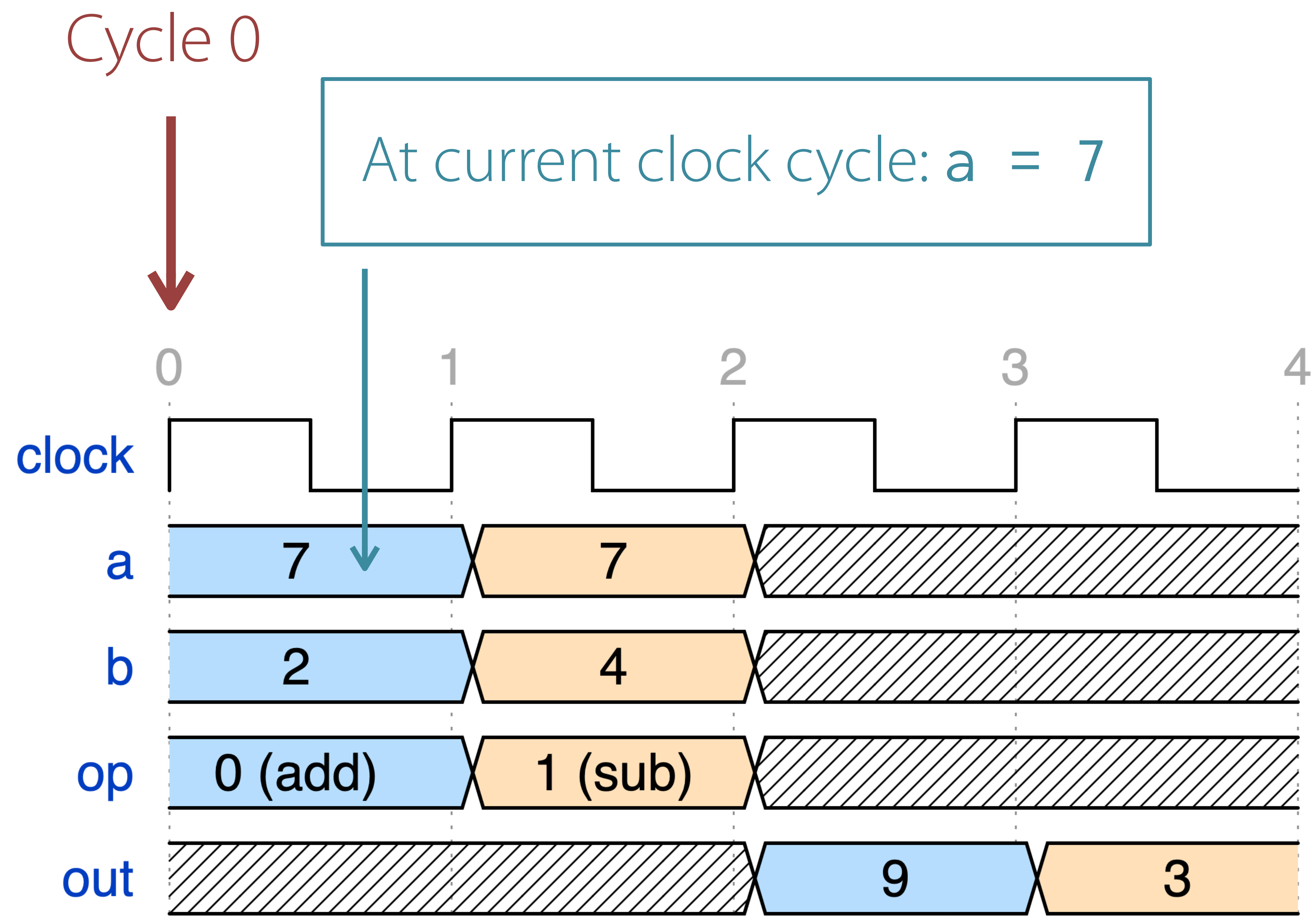
```
prot sub(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 1;
  step();
  …
```

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();
  …
```

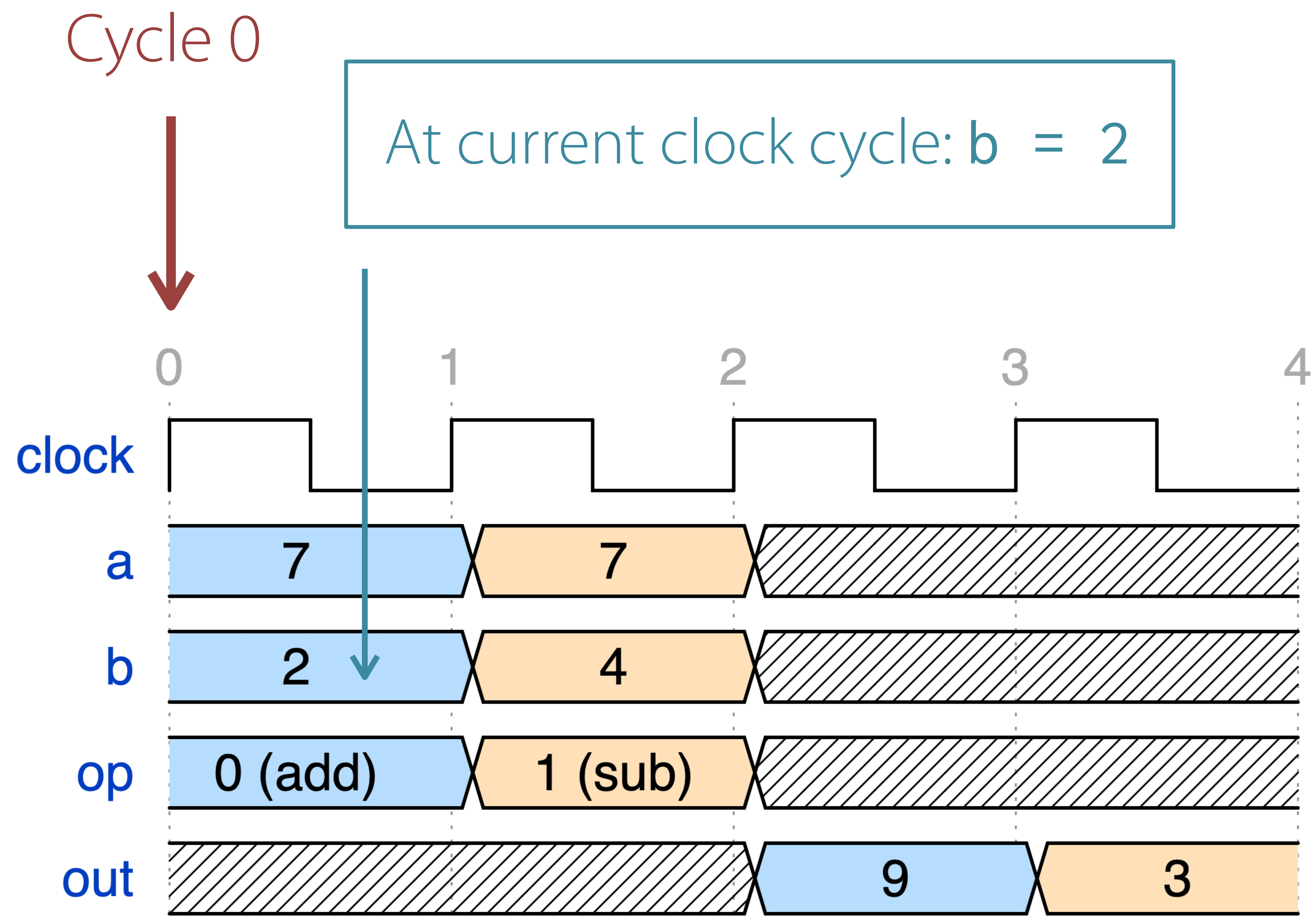
add(a=7, b=2, out=?)

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();
  …
```

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

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

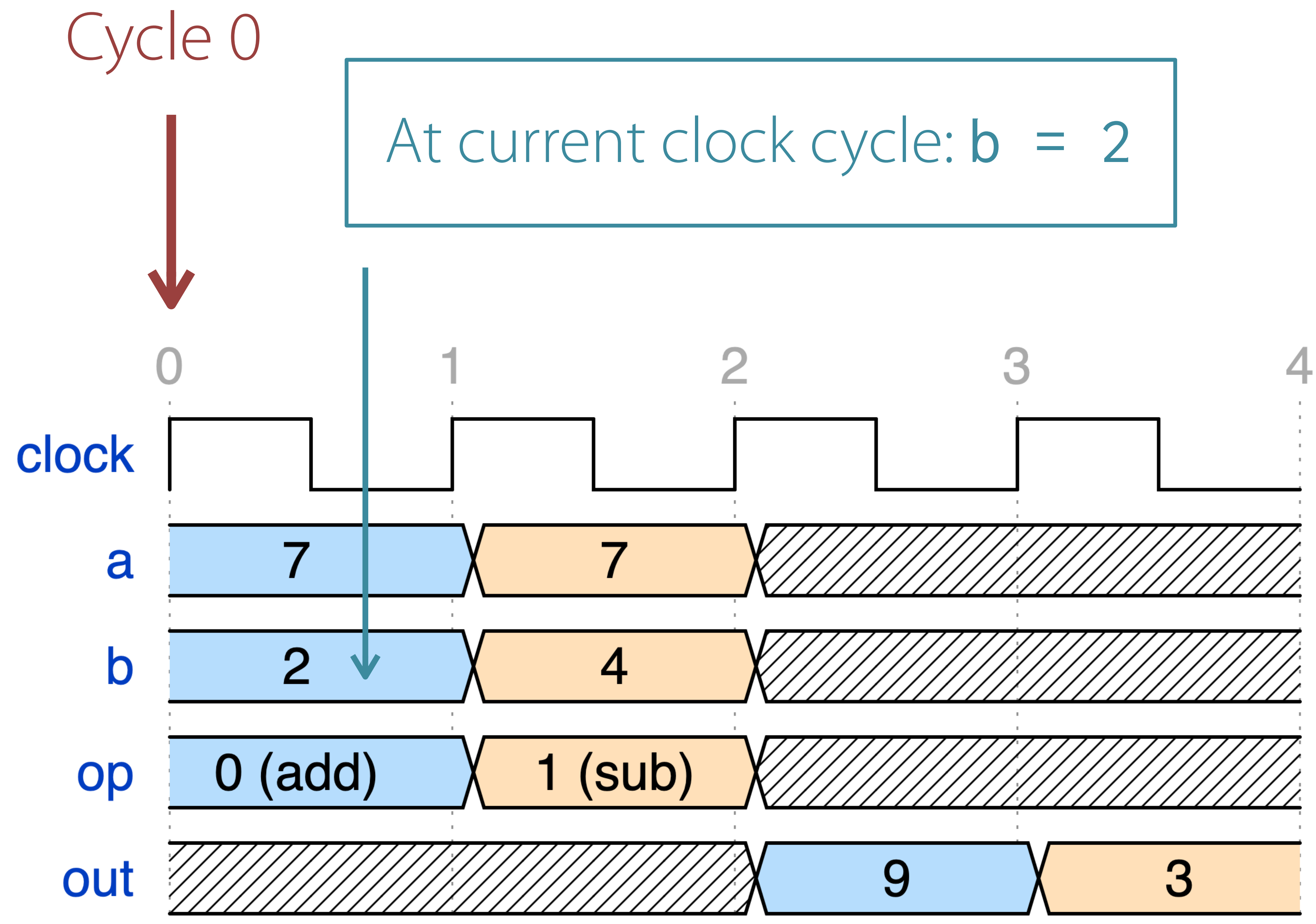
sub(a=7, b=?, out=?)

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



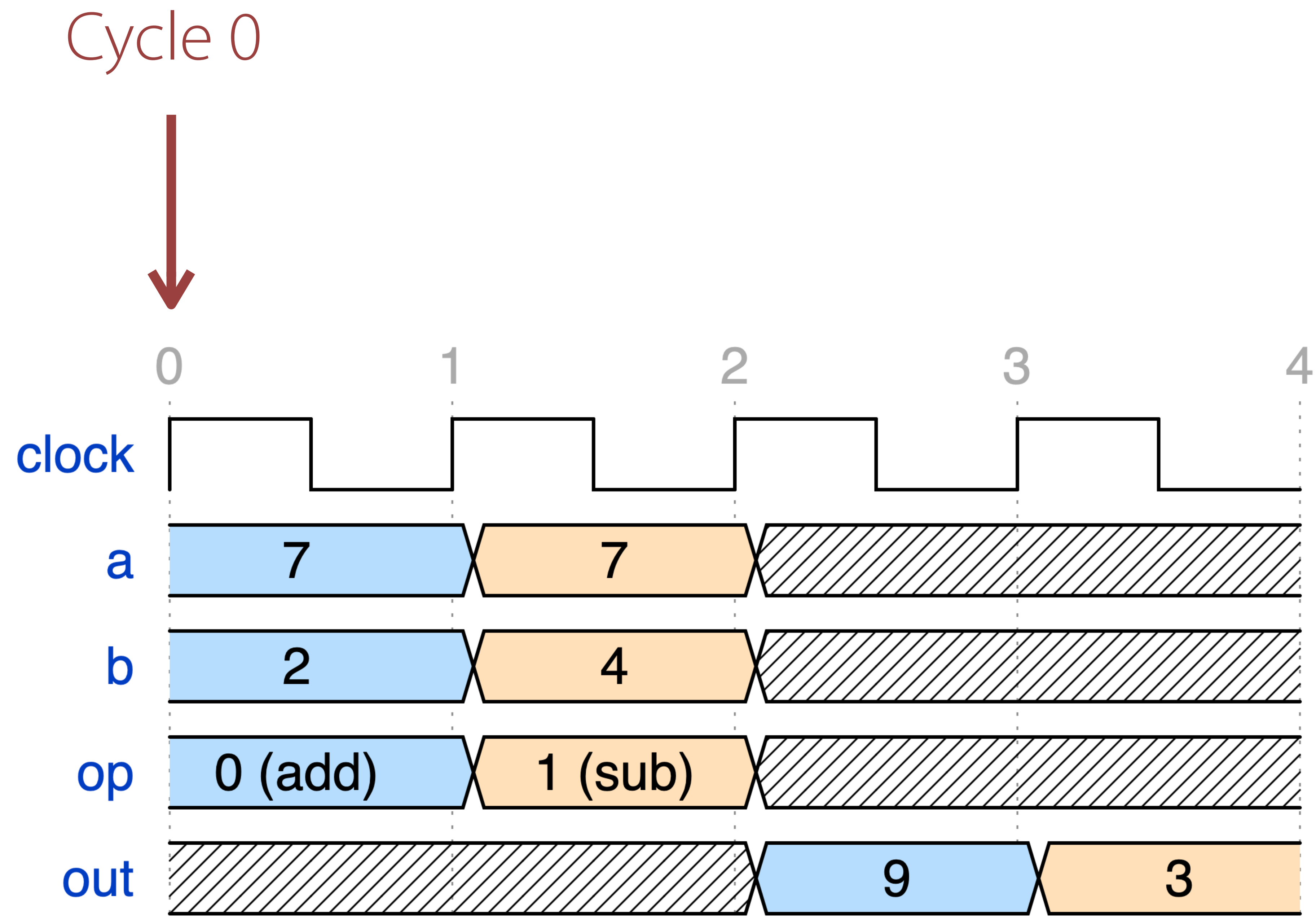
add(a=7, b=2, out=?)

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

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

Constraints: { a = 7, b = 2 }

New!



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=?)

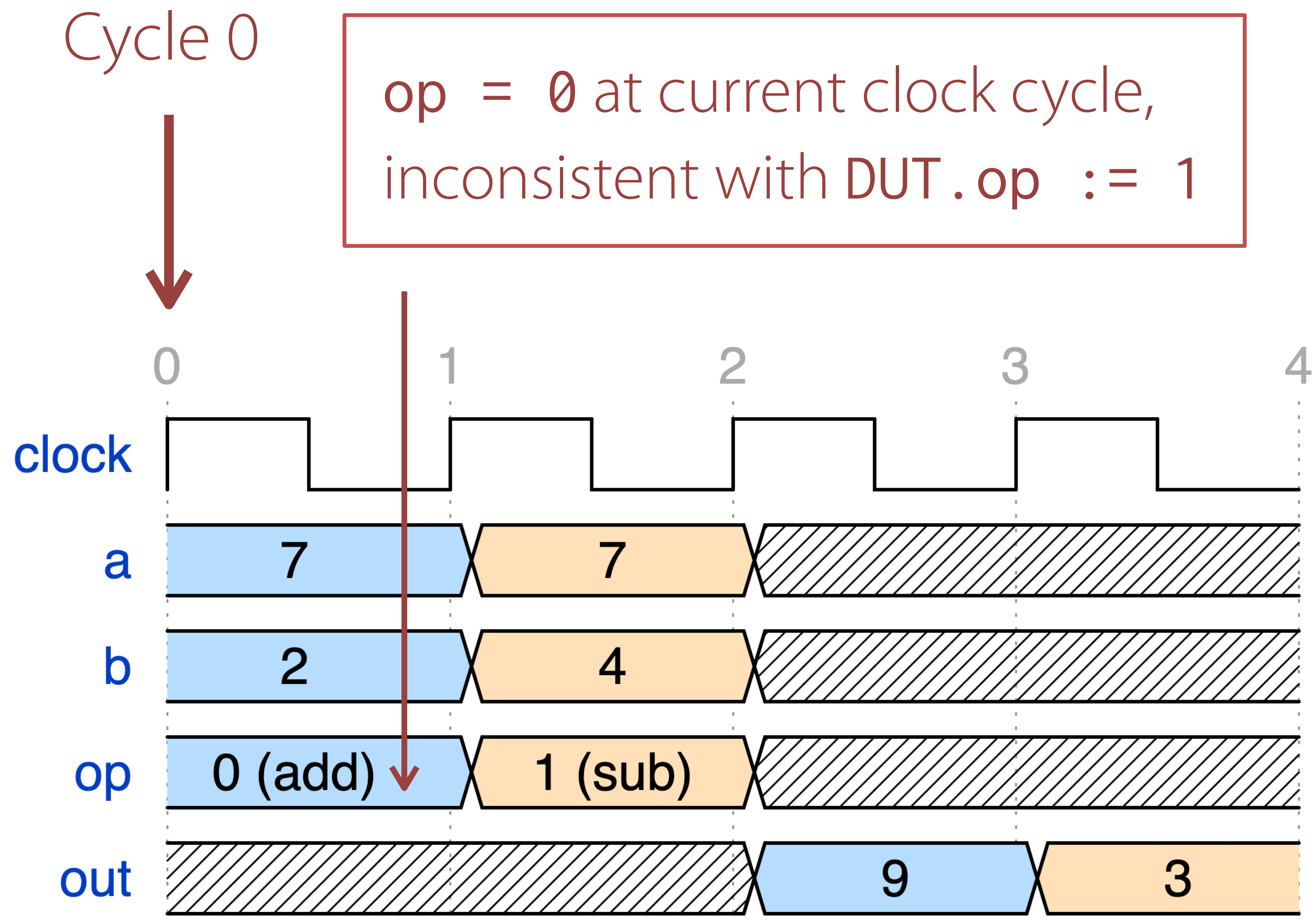
Constraints: { a = 7, b = 2 }

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`

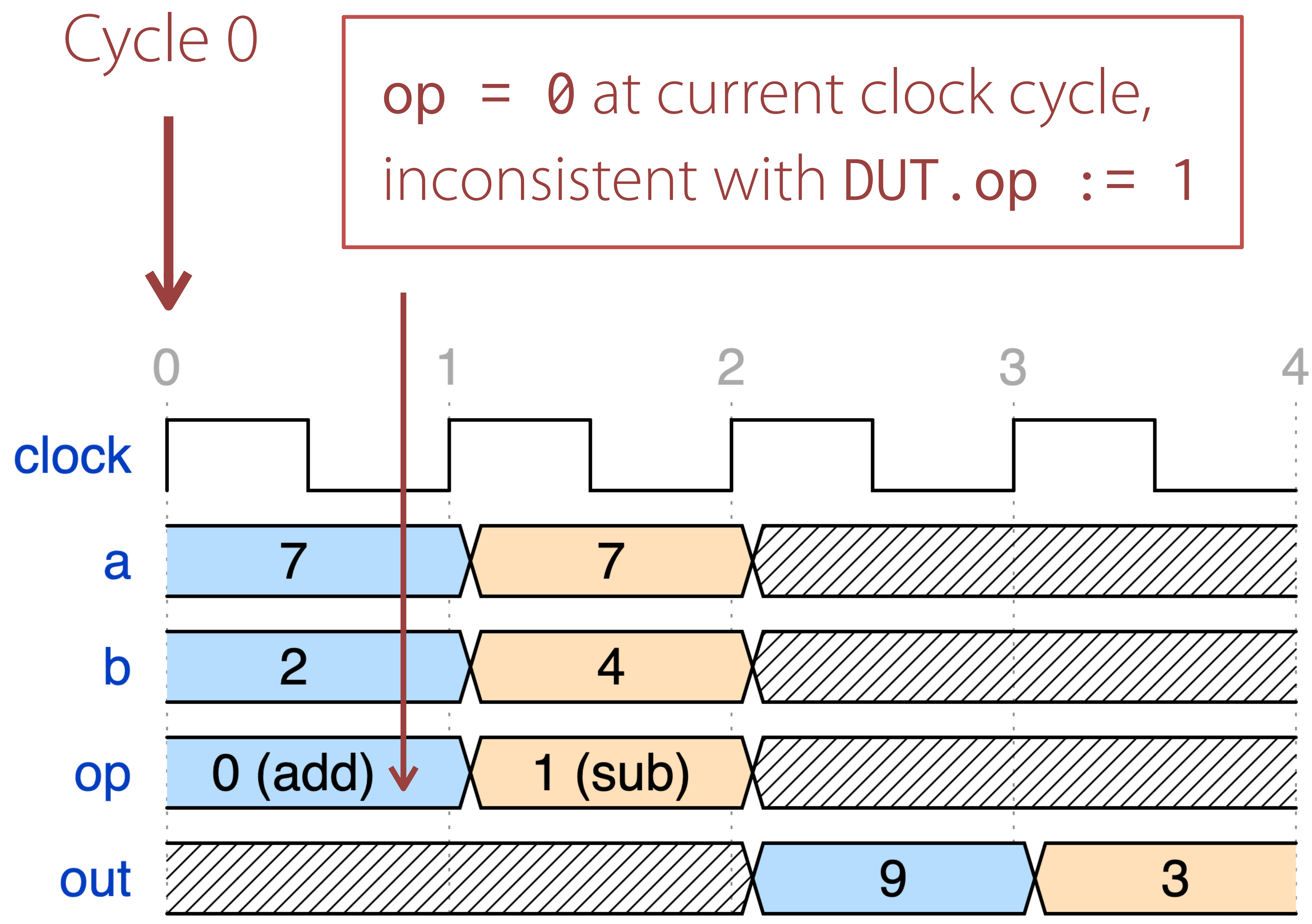


`add(a=7, b=2, out=?)`

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

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

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



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=?)

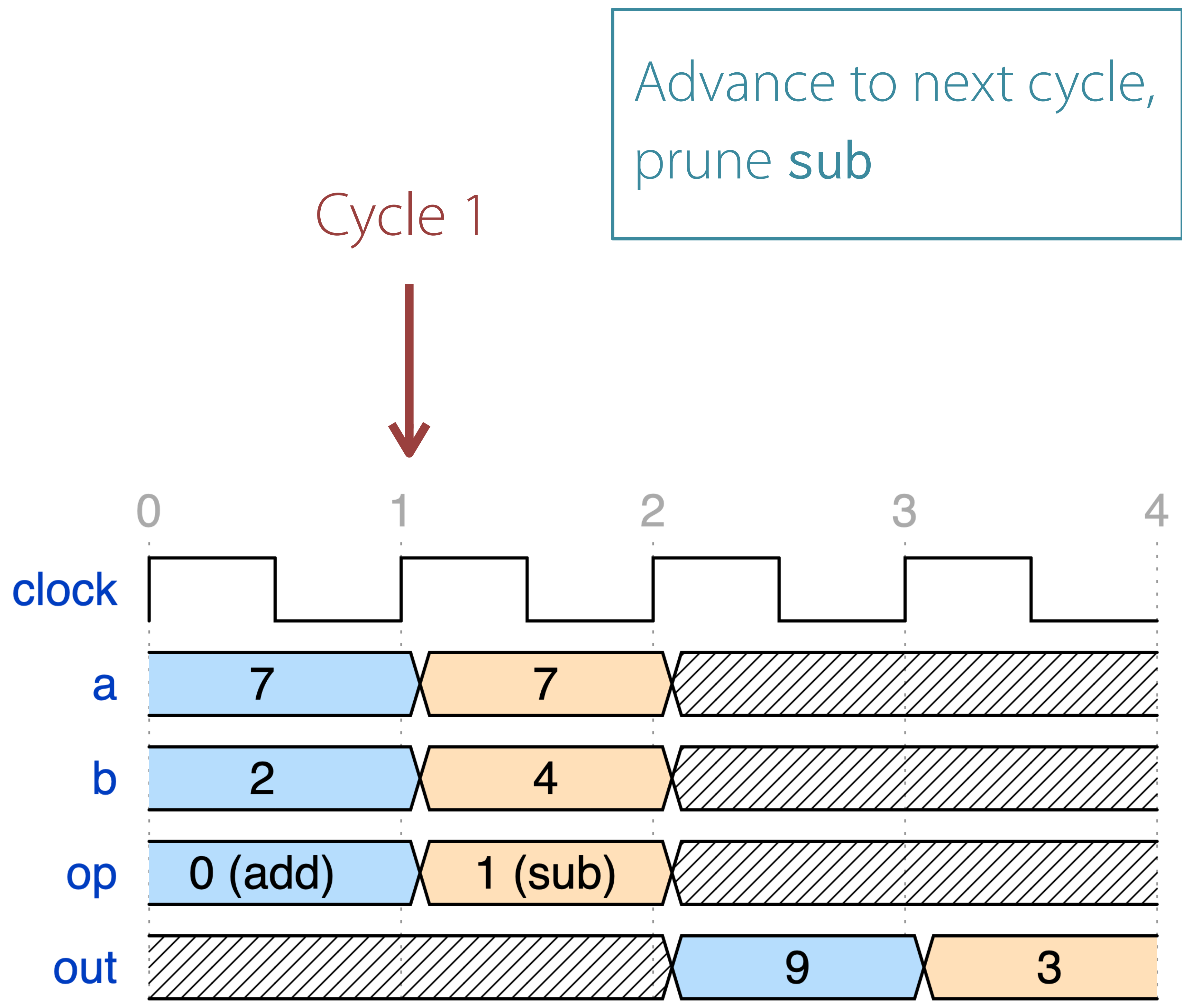
Constraints: { a = 7, b = 2, 0 = 0 }

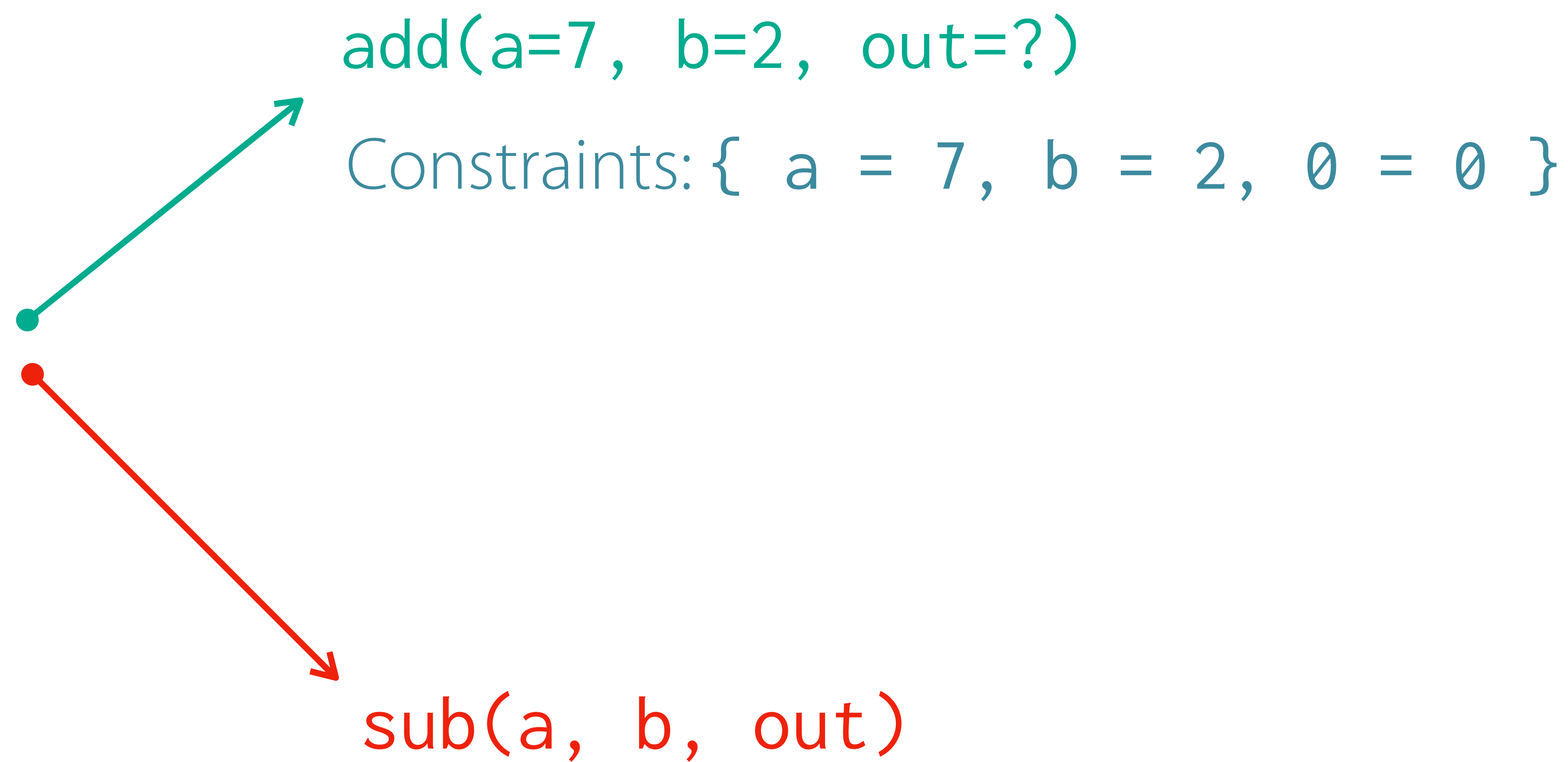
sub(a, b, out)

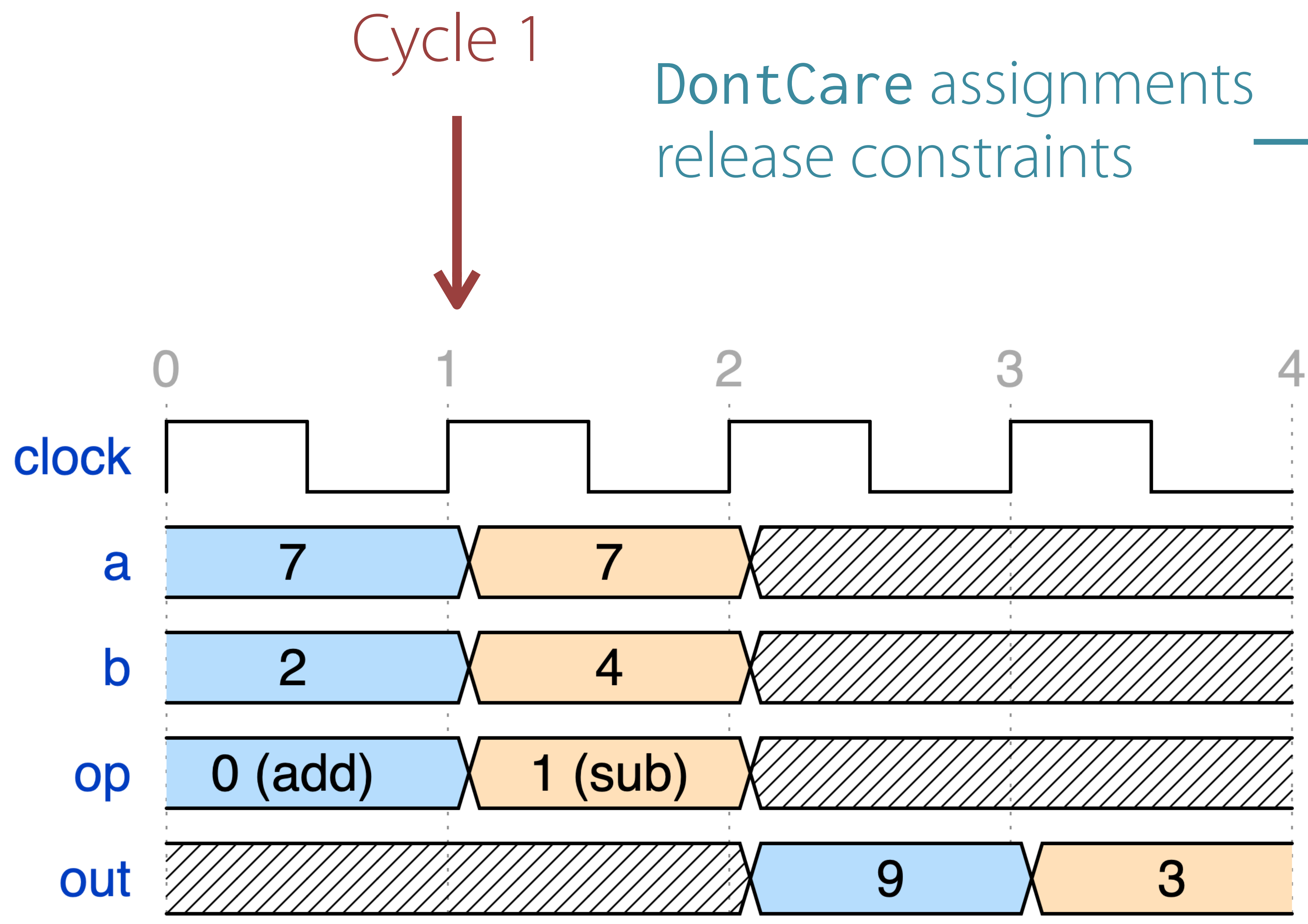
X Can't have occurred during cycle 0! This path is pruned

```
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();
  …
```

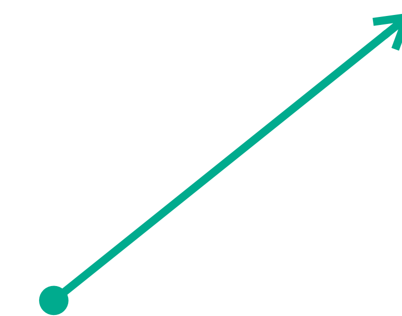






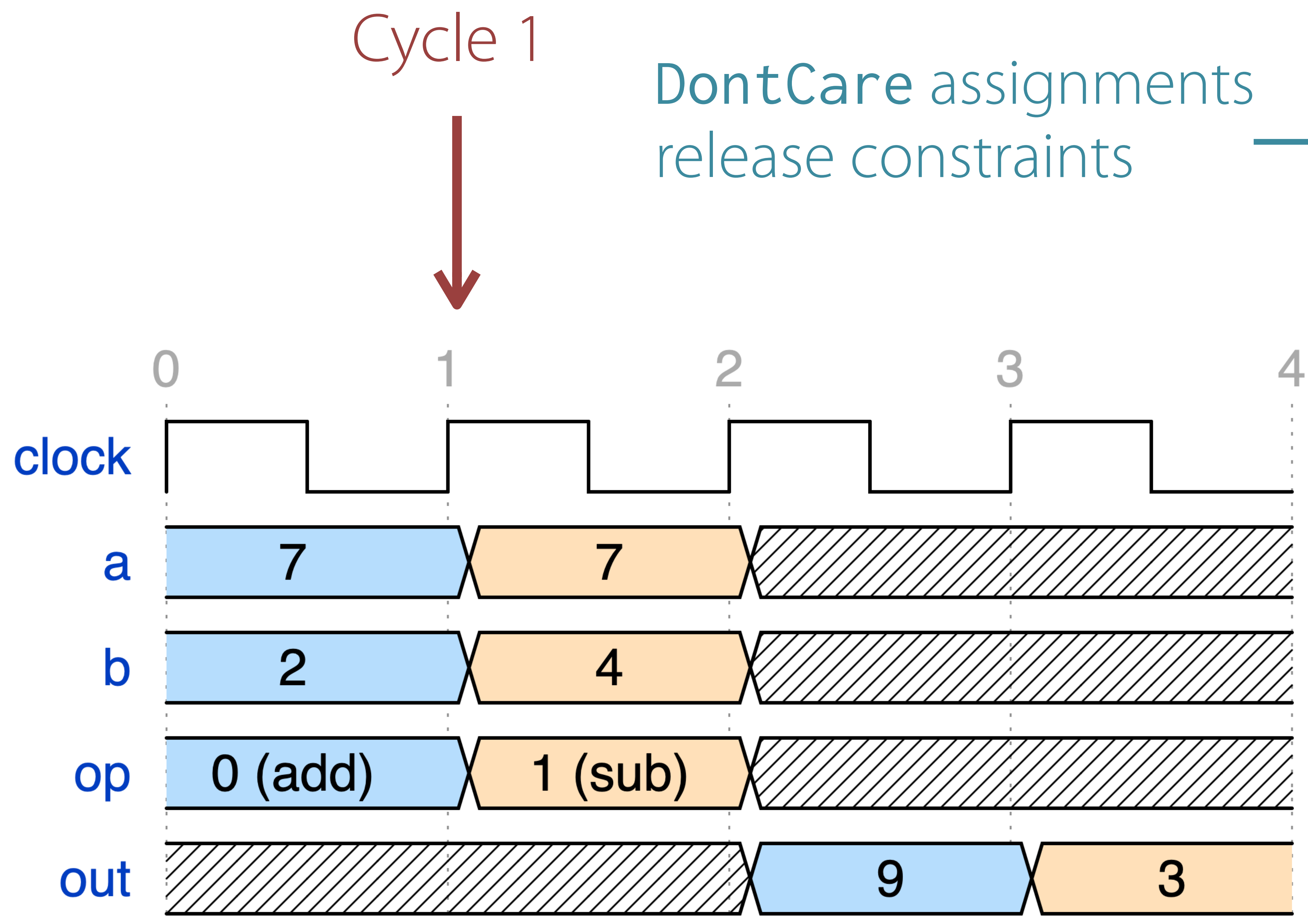
DontCare assignments release constraints

```
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 }`



DontCare assignments release constraints

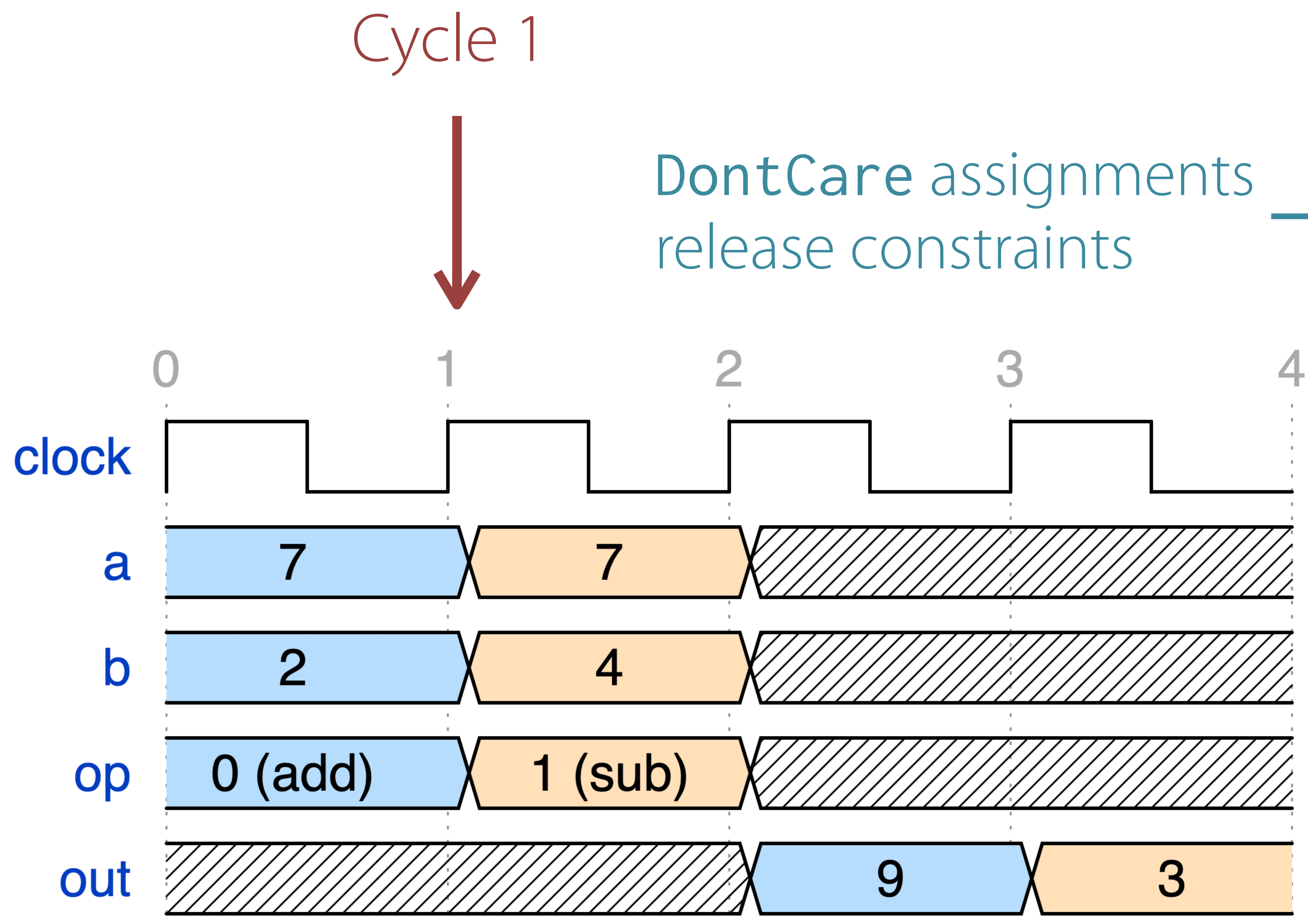
```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

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

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

Constraint `a = 7` removed

Cycle 1

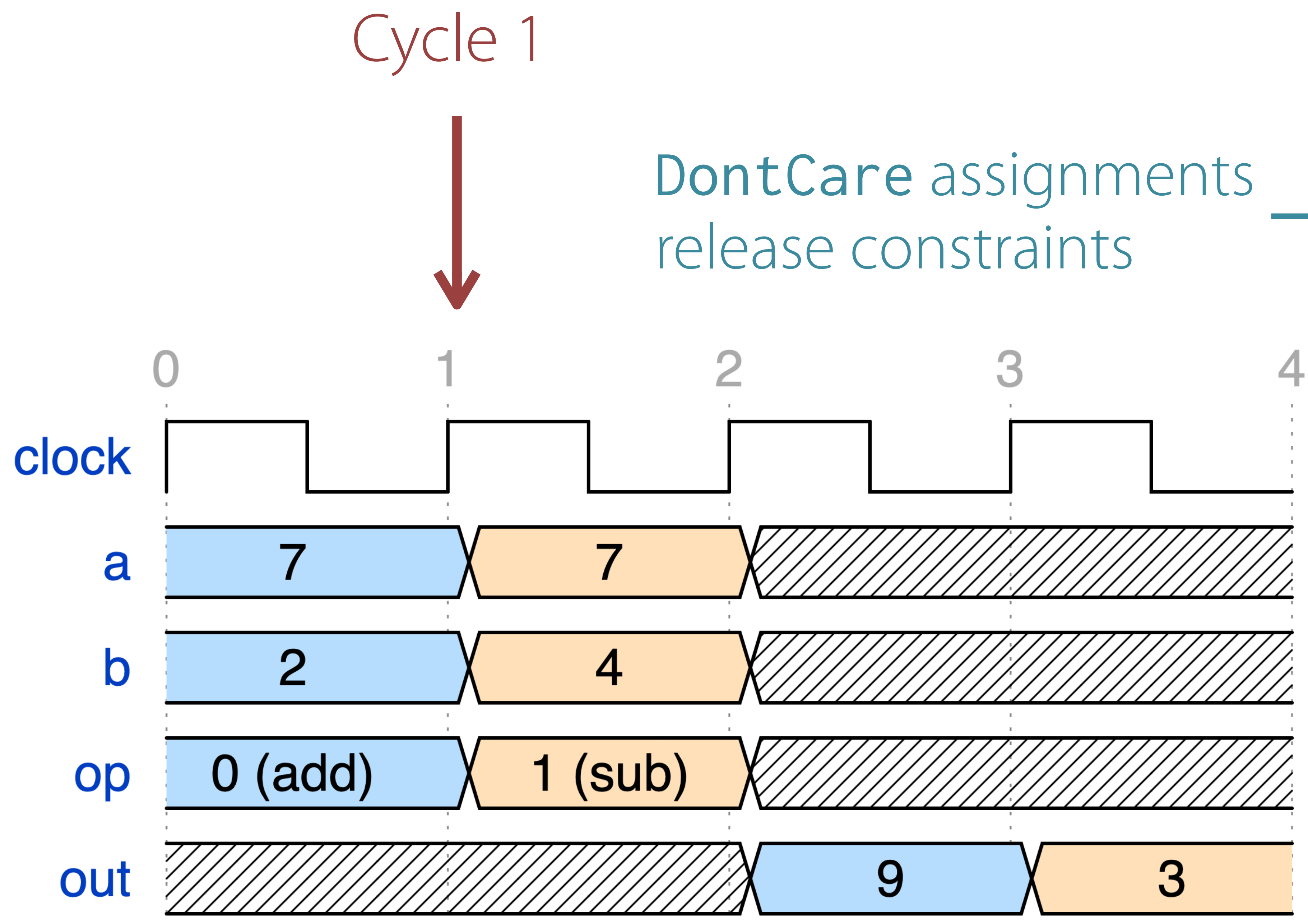


DontCare assignments release constraints

```
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 }



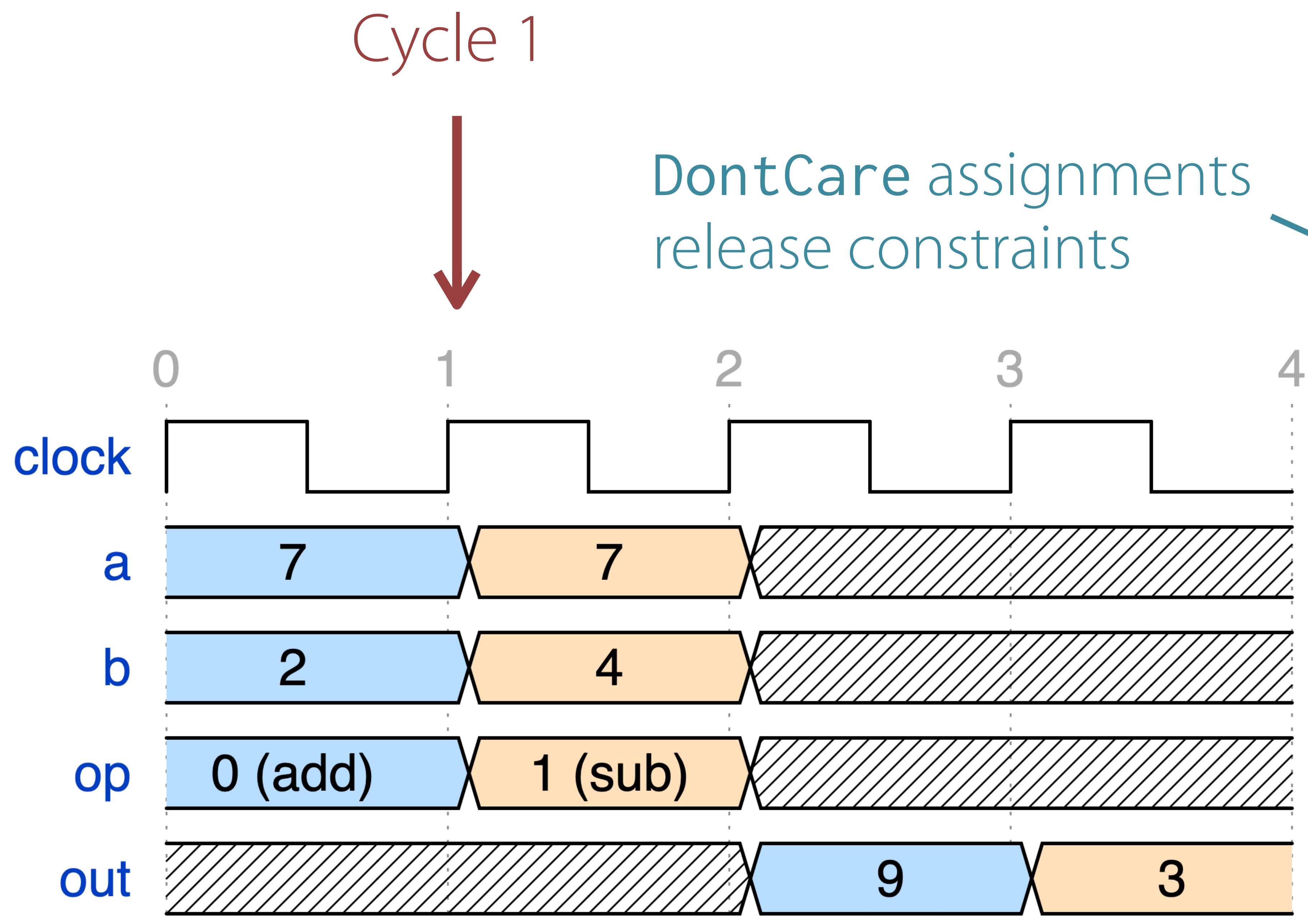
`DontCare` assignments
release constraints

```
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

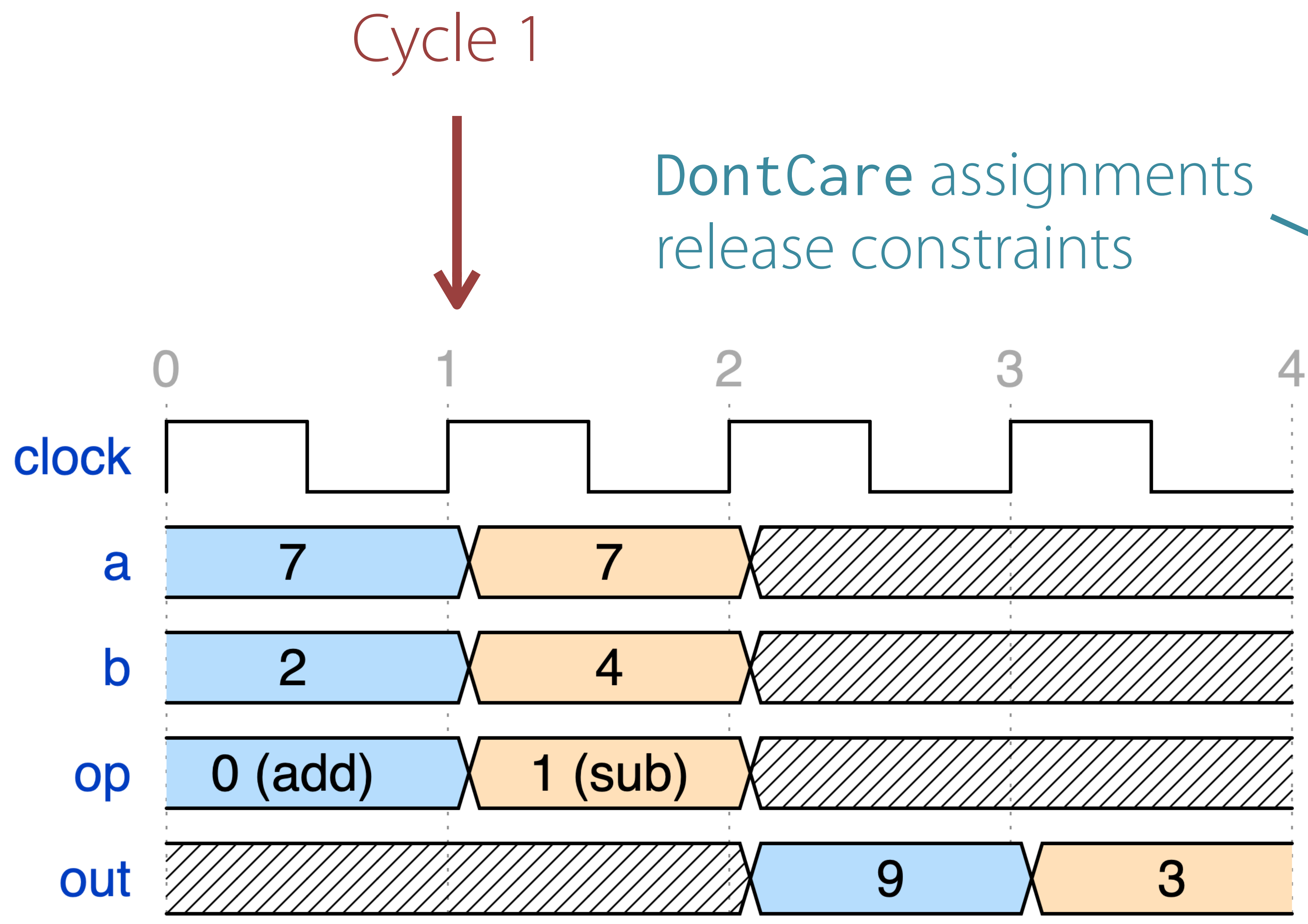


DontCare assignments release constraints

```
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 }



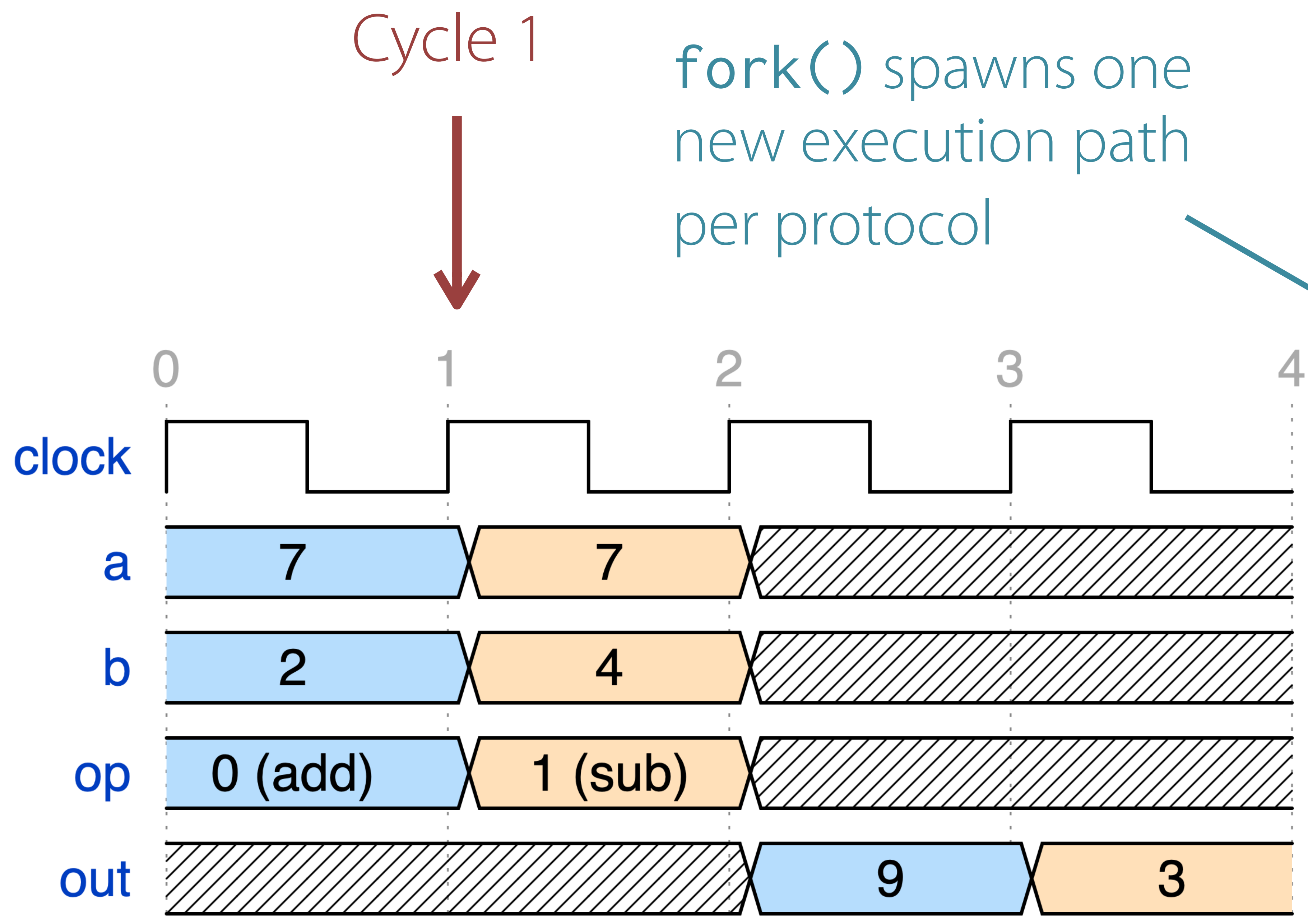
DontCare assignments release constraints

```
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: ∅

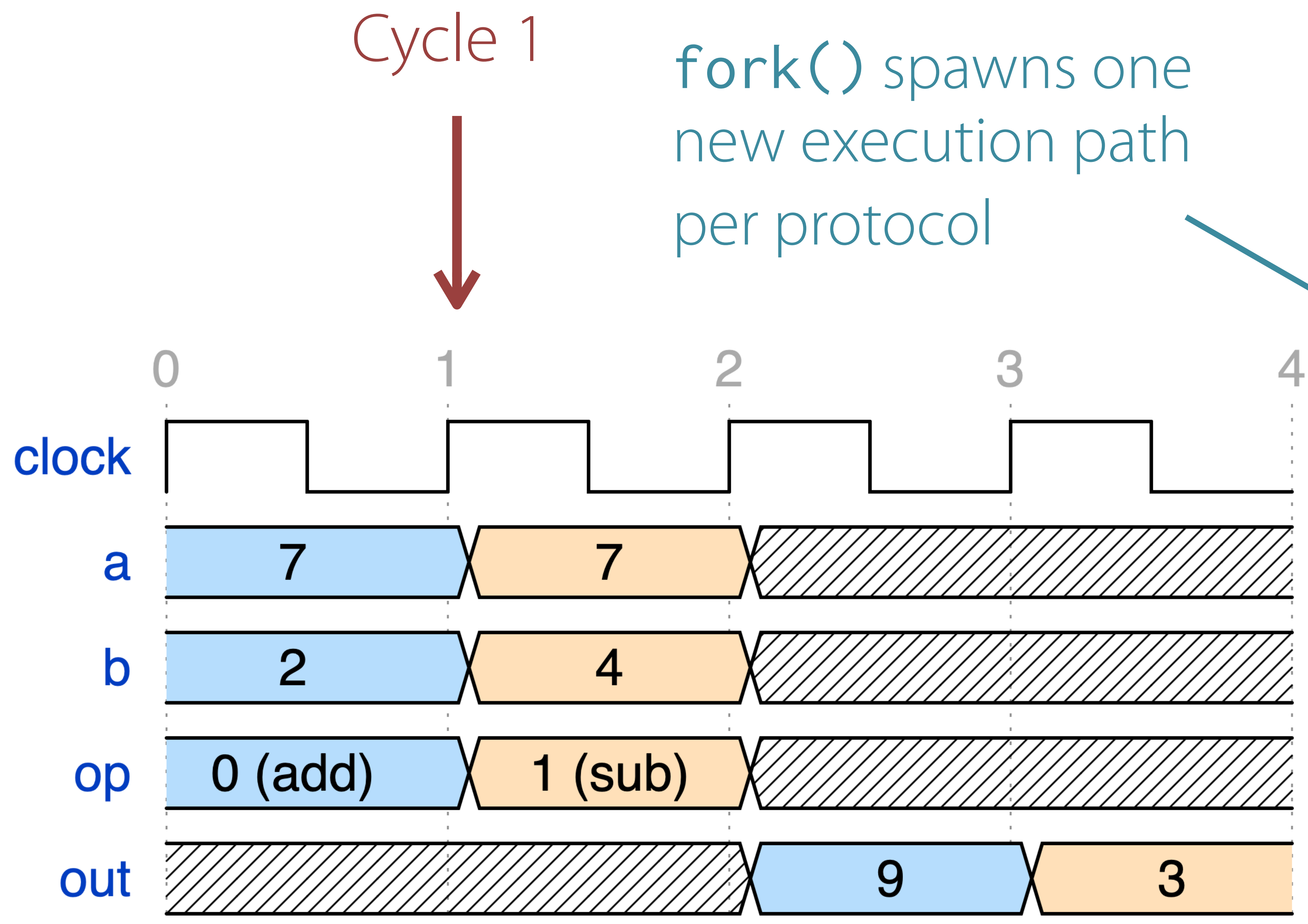
Constraint `0 = 0` removed



fork() spawns one new execution path per protocol

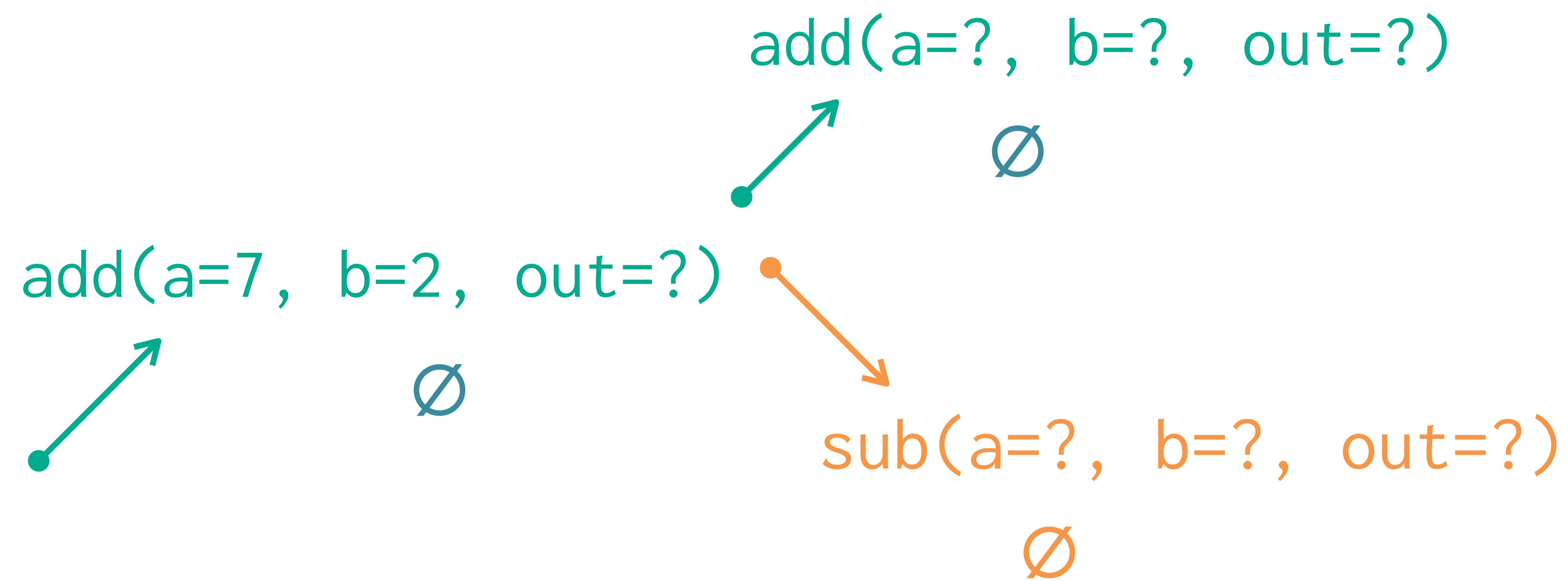
```
prot add<DUT: ALU>(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X;
  DUT.b := X;
  DUT.op := X;
  fork();
  step();
  assert_eq(DUT.out, out);
}
```

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

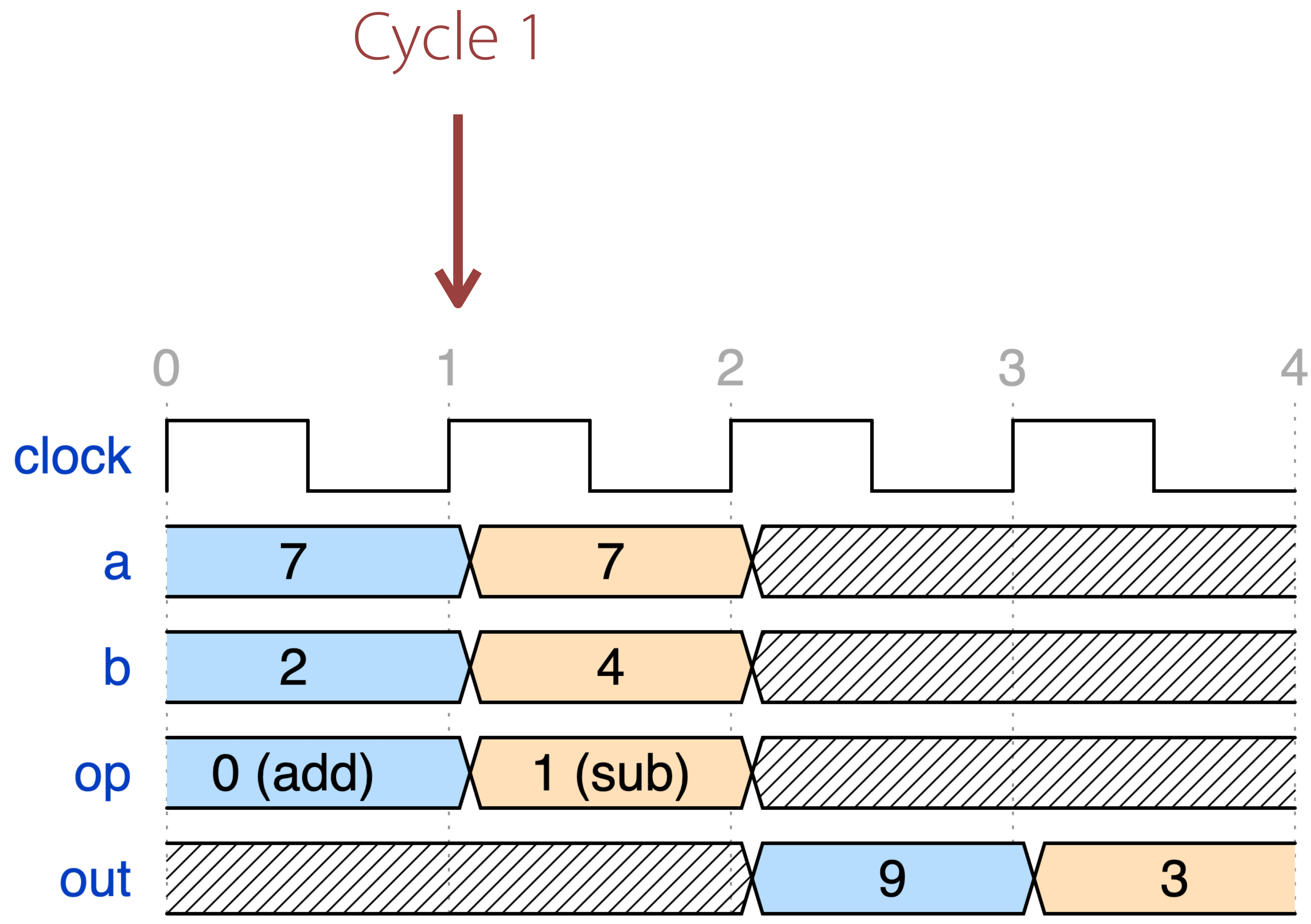


`fork()` spawns one new execution path per protocol

```
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);
}
```

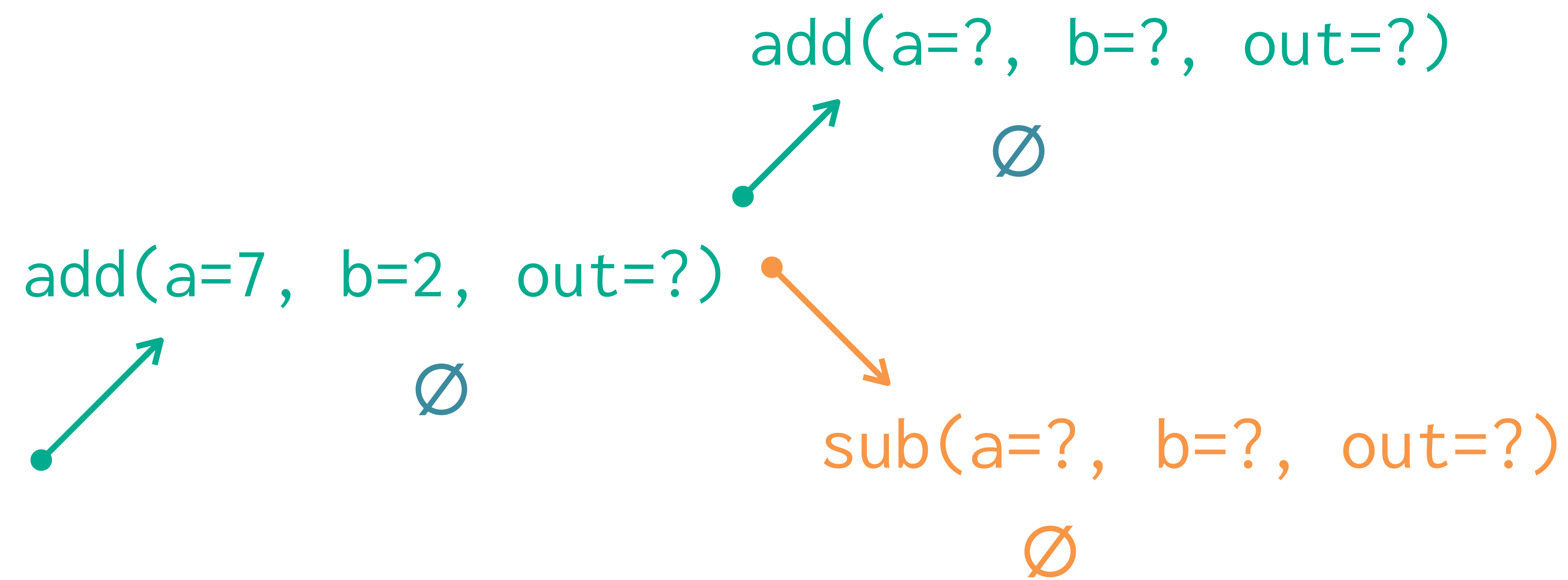


Skipping ahead…

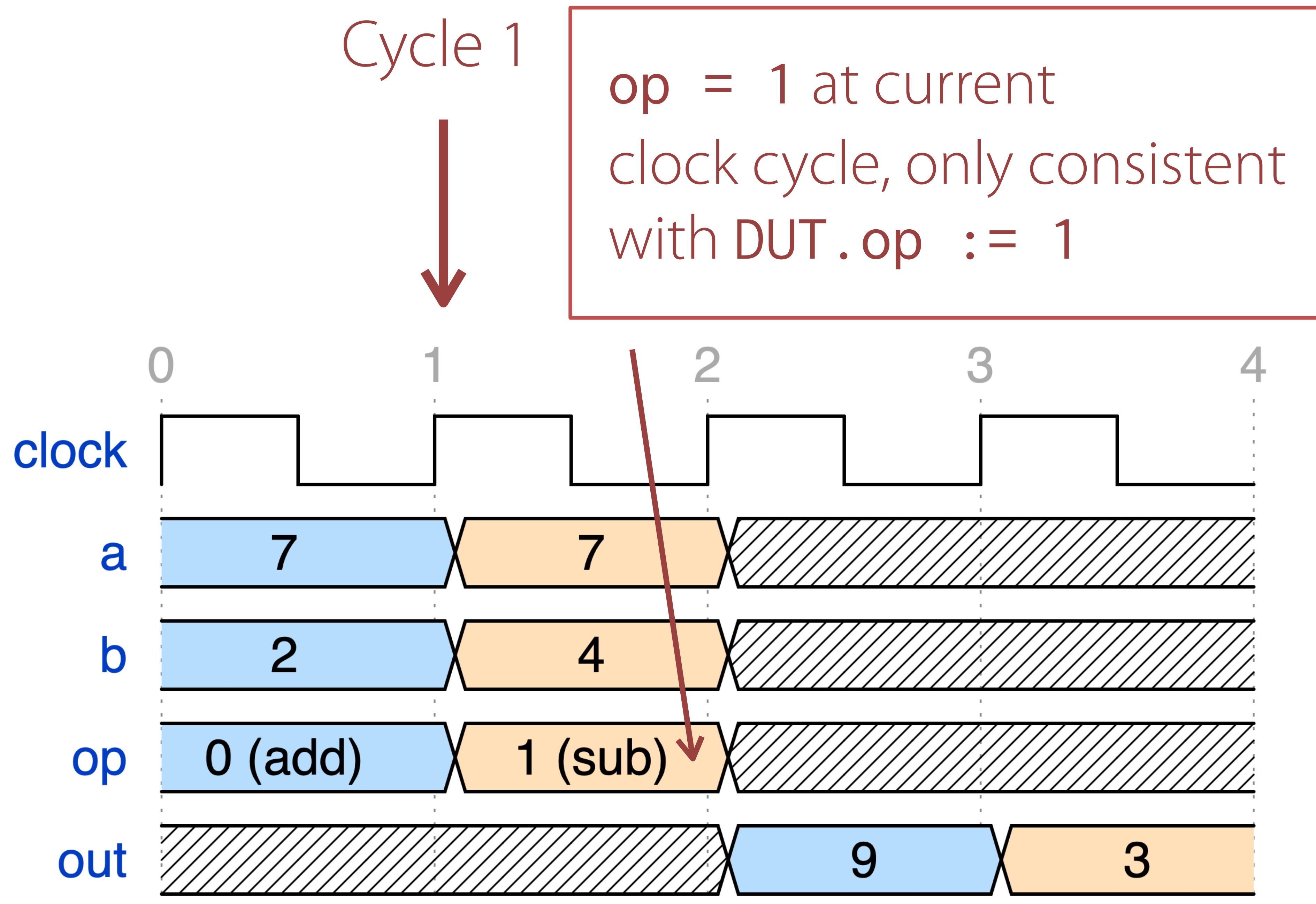



```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

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

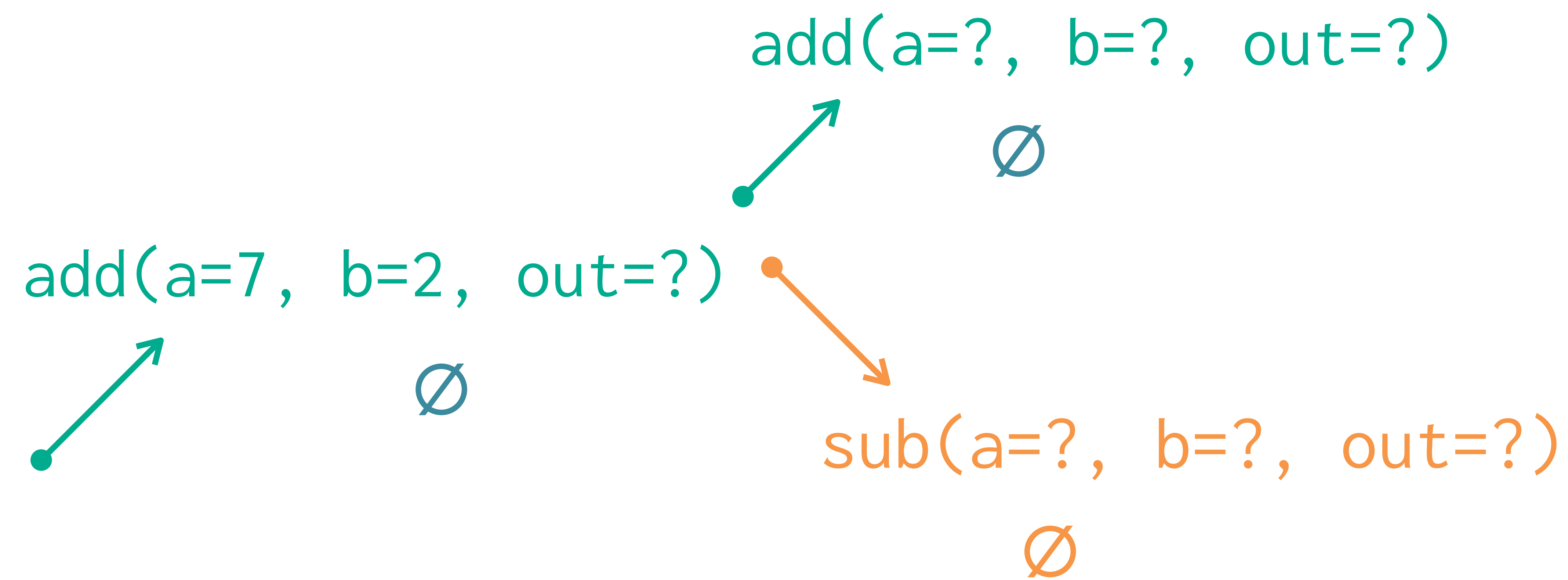


Skipping ahead…

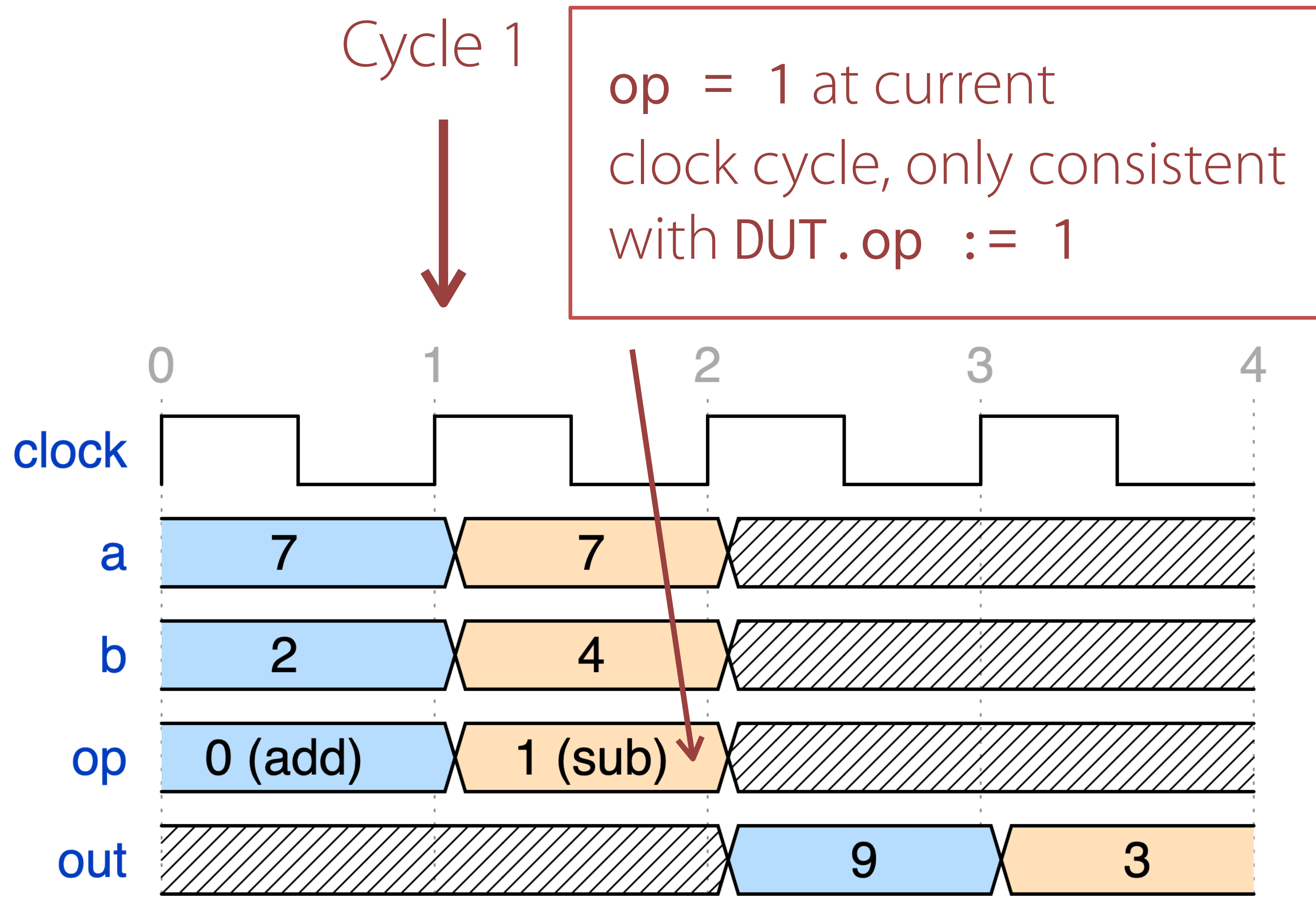



```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

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



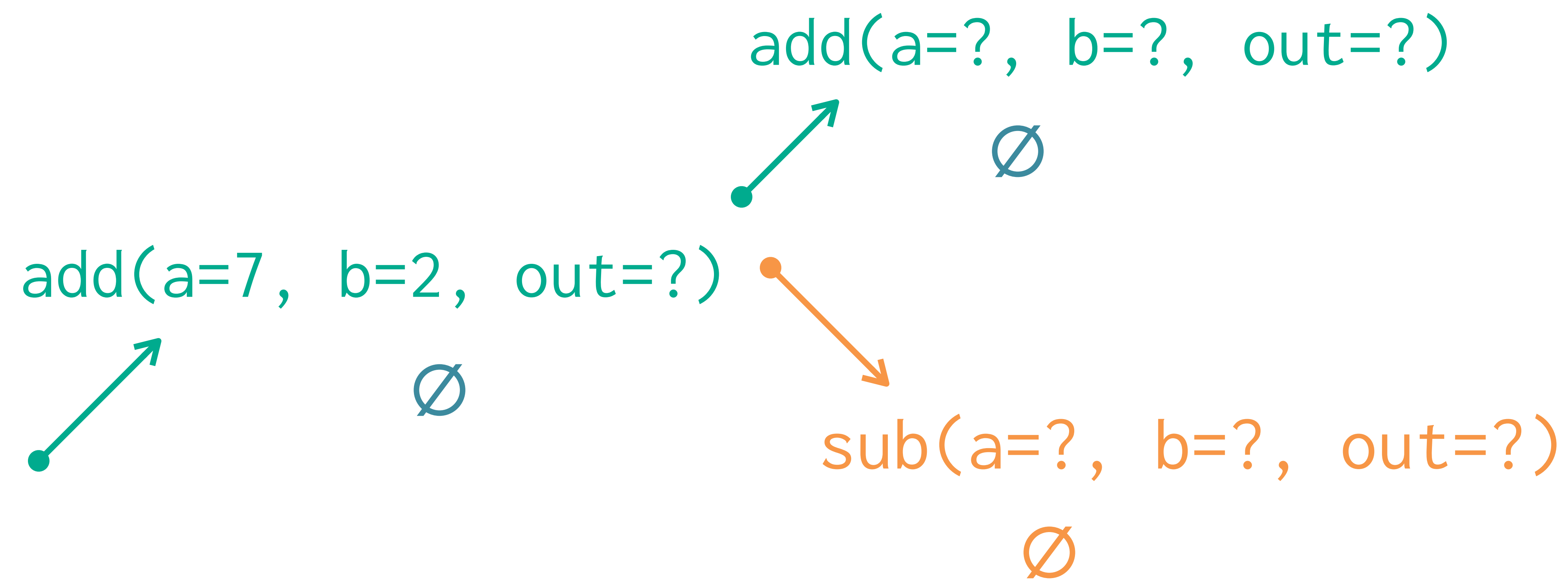
Skipping ahead…



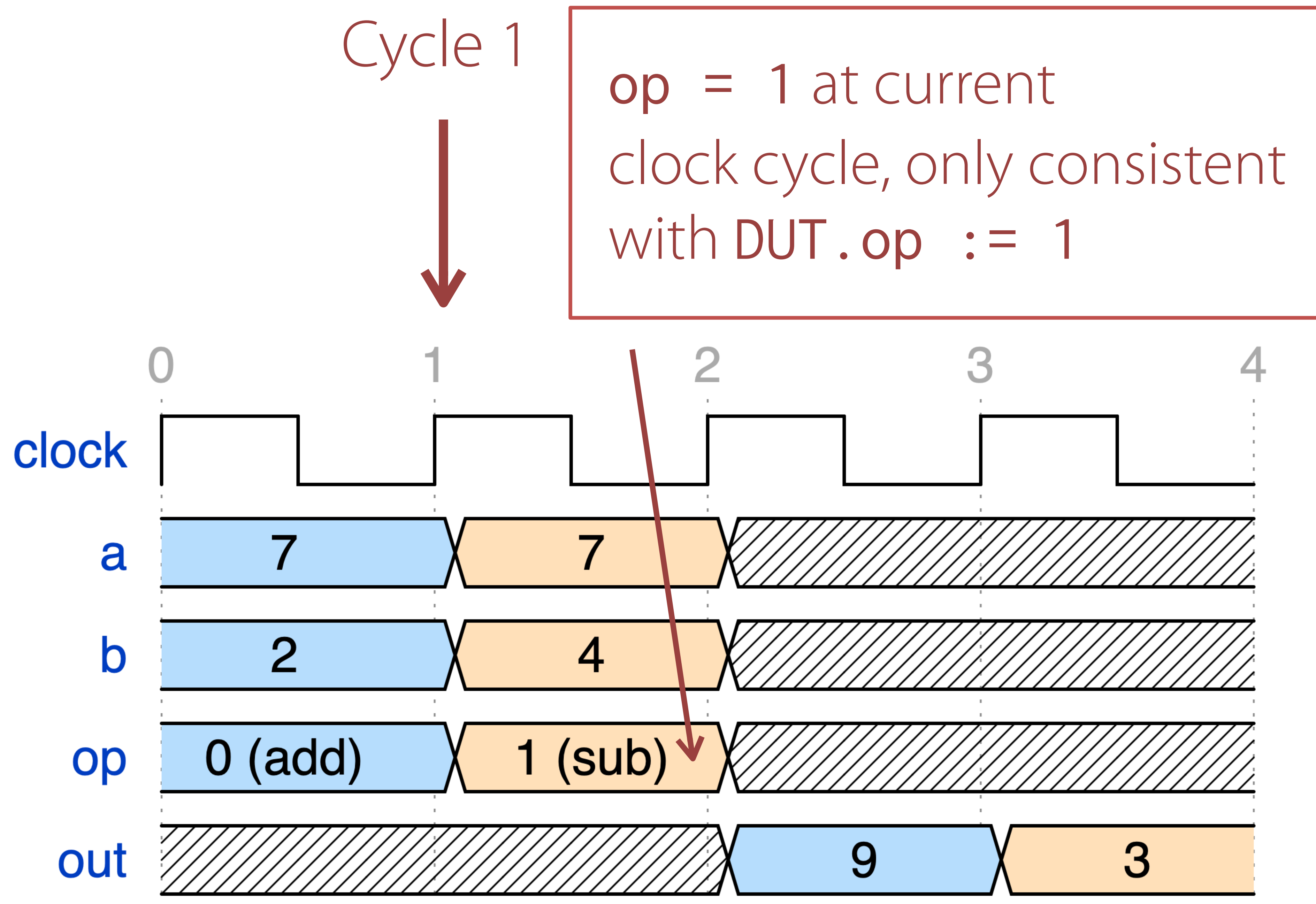

```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

Inconsistent!

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



Skipping ahead…

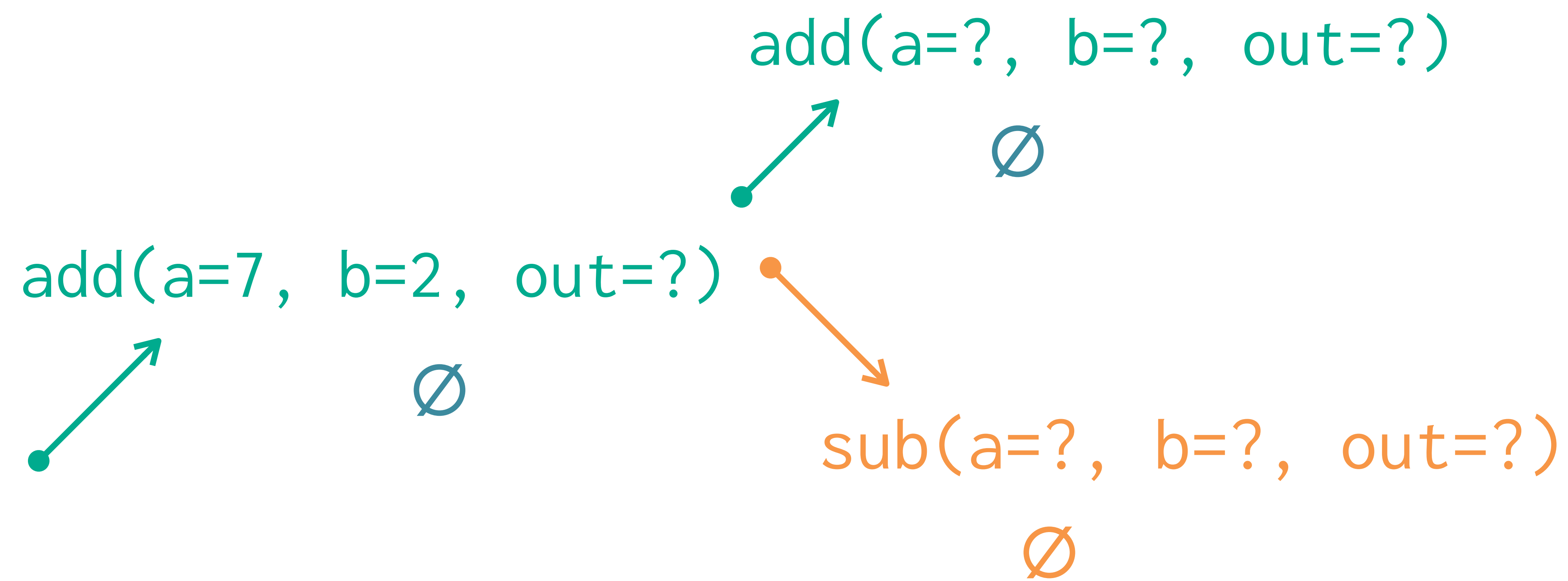



```
prot add(a, b, out) {
  DUT.a := a;
  DUT.b := b;
  DUT.op := 0;
  step();
  …
```

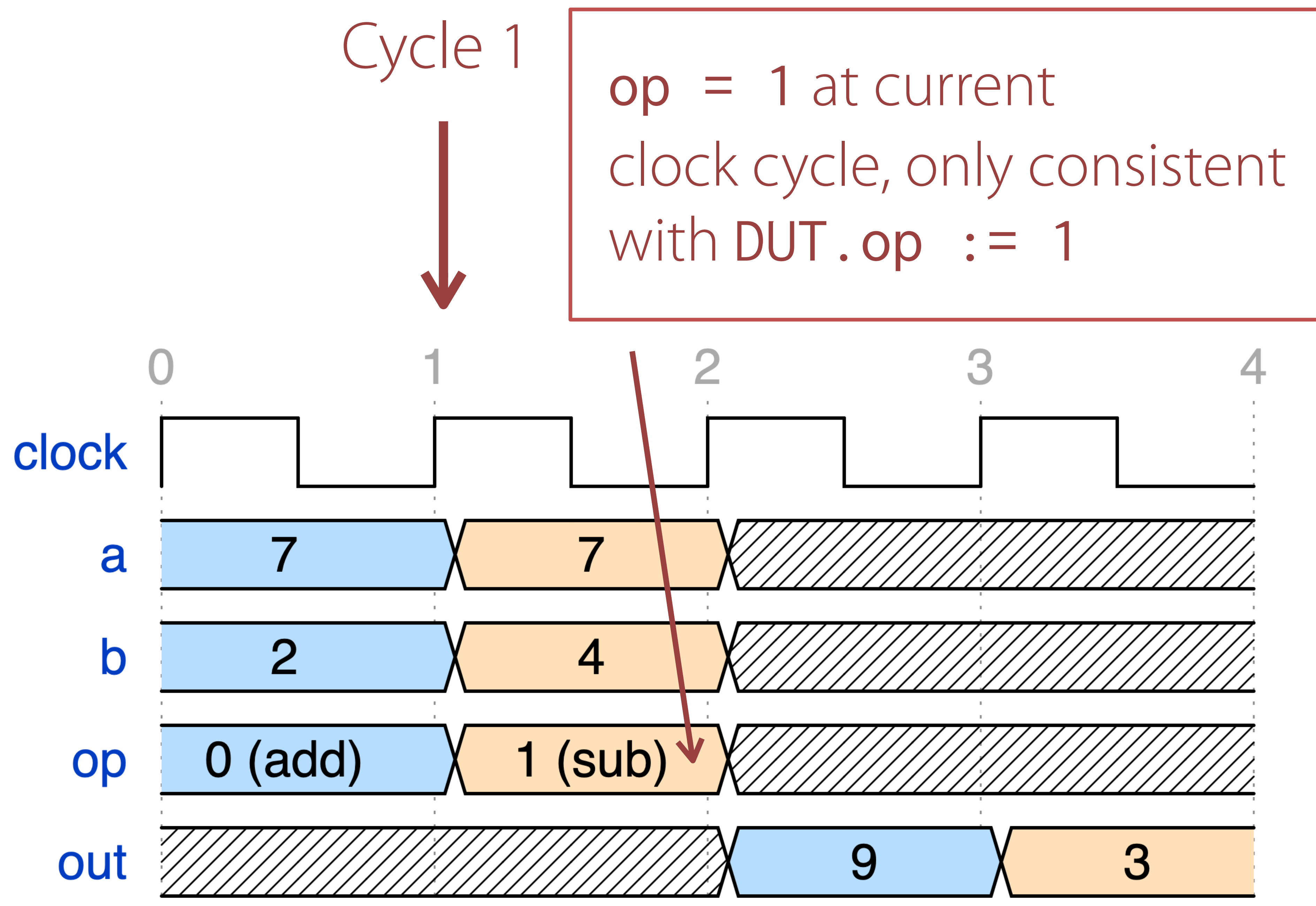
Inconsistent!

```
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();
  …
```

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

Inconsistent!

Consistent

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

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

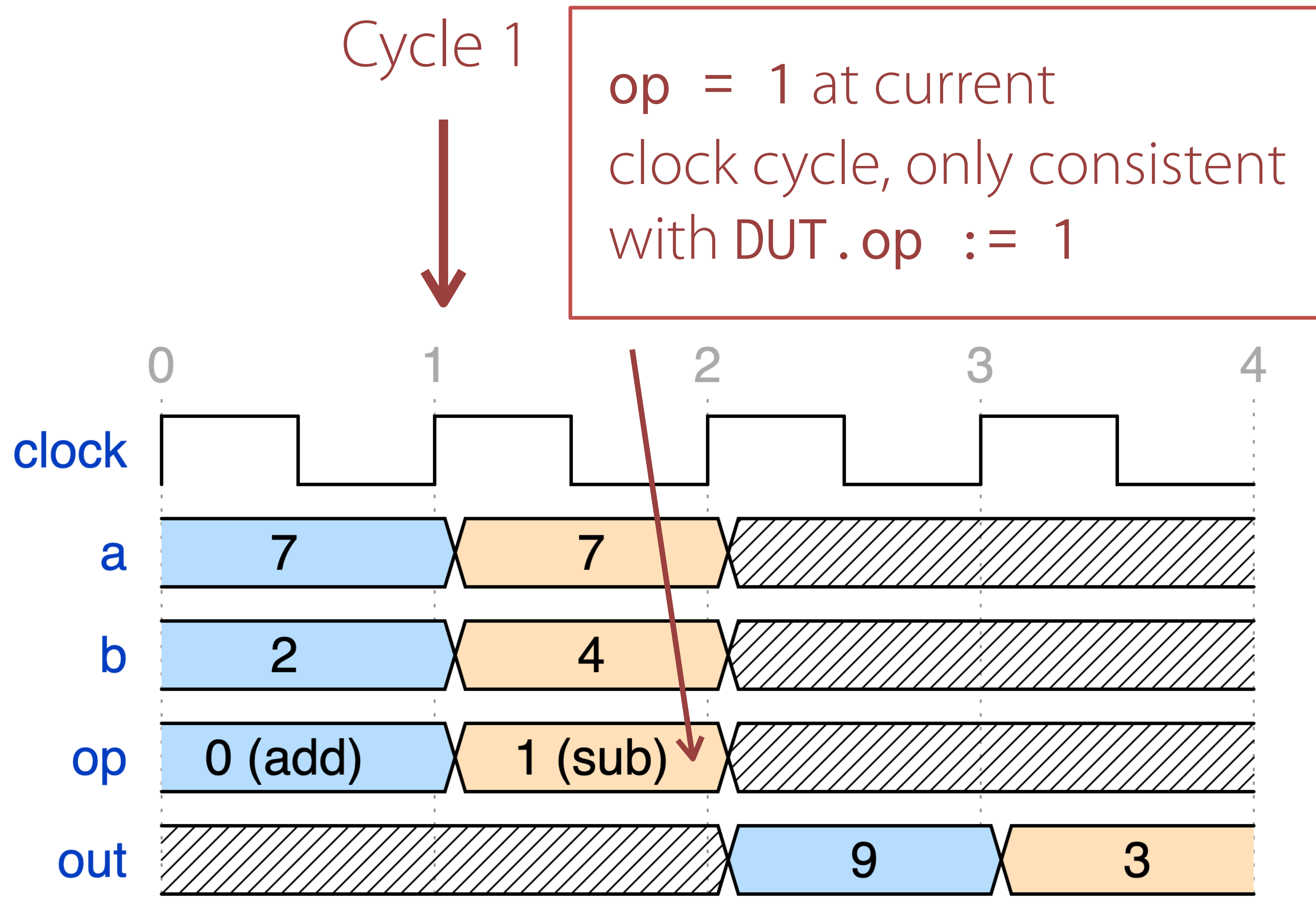
add(a=7, b=2, out=?)

∅

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

∅

Skipping ahead…



op = 1 at current clock cycle, only consistent with `DUT.op := 1`

```
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

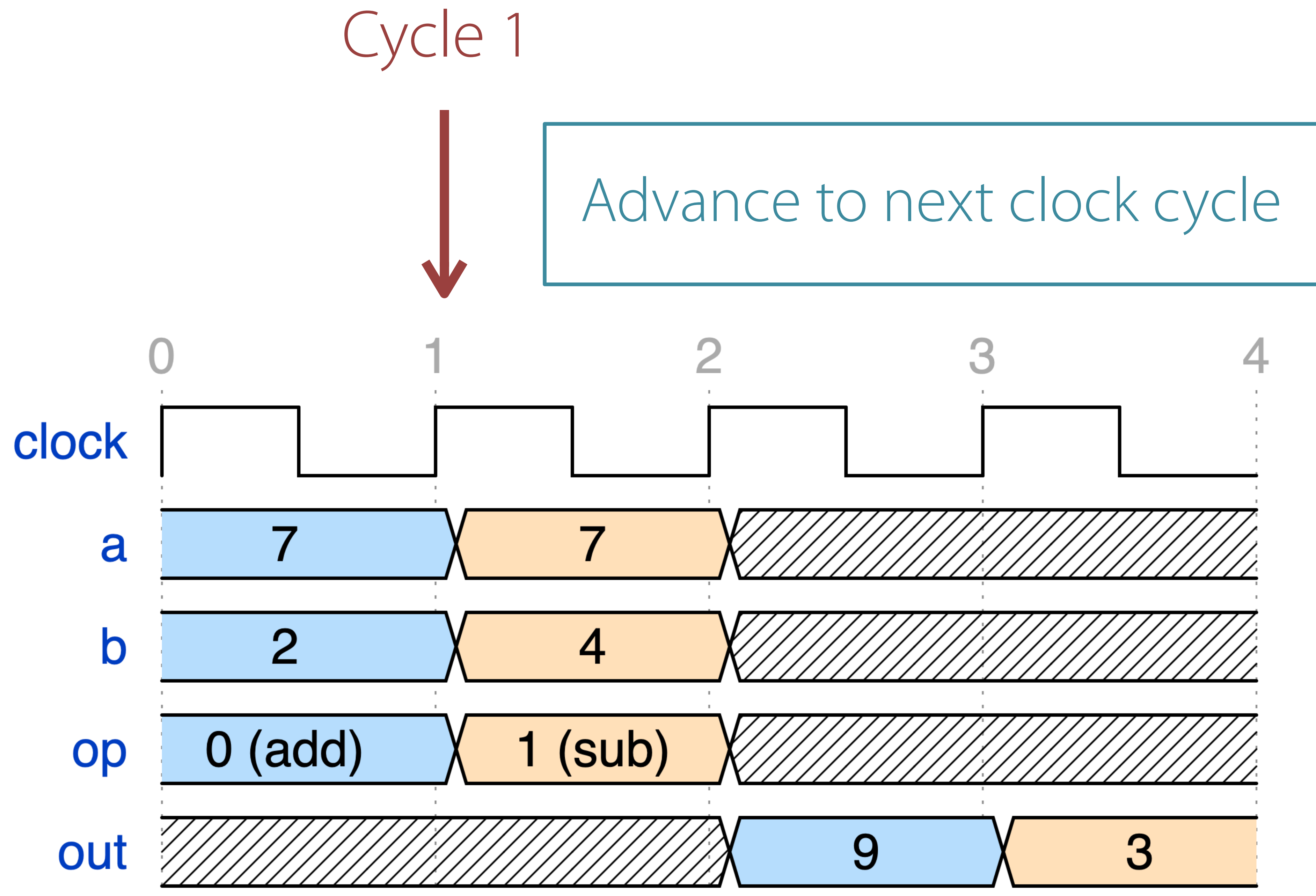
`add(a=7, b=2, out=?)`

∅

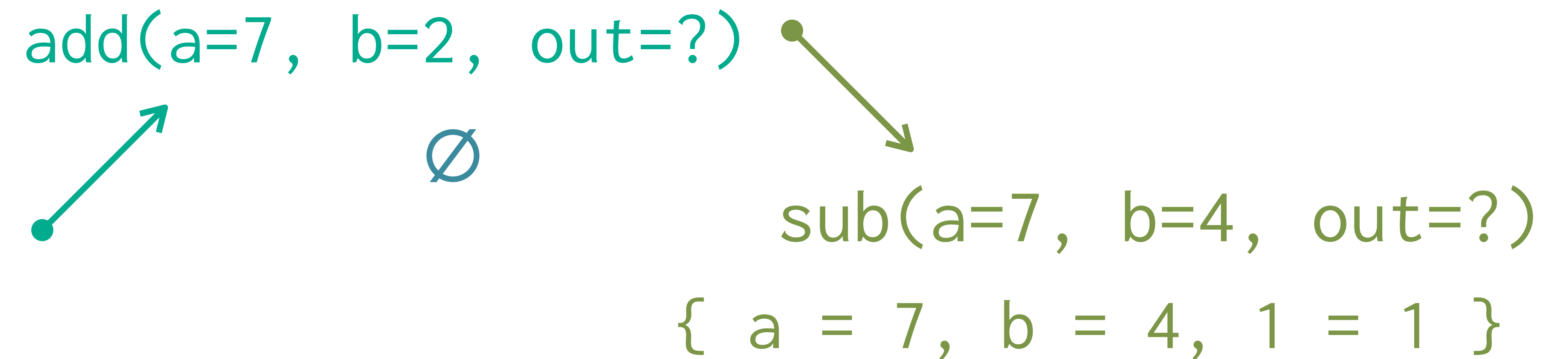
`sub(a=7, b=4, out=?)` ✓

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

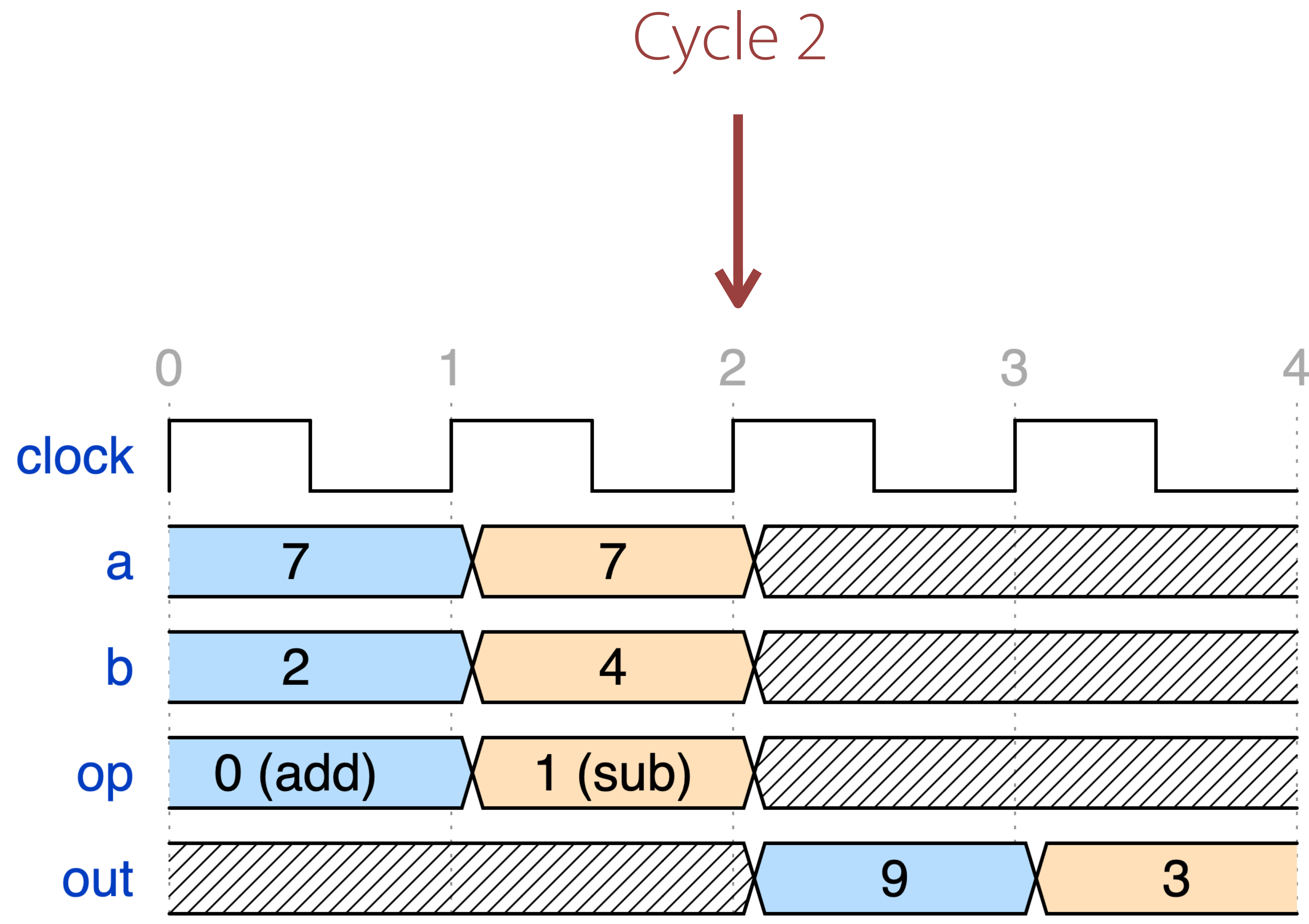
Parent add transaction continues…




```
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);
}
```

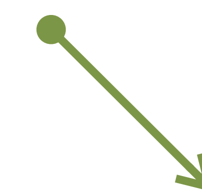


Parent add transaction continues…




```
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);
}
```

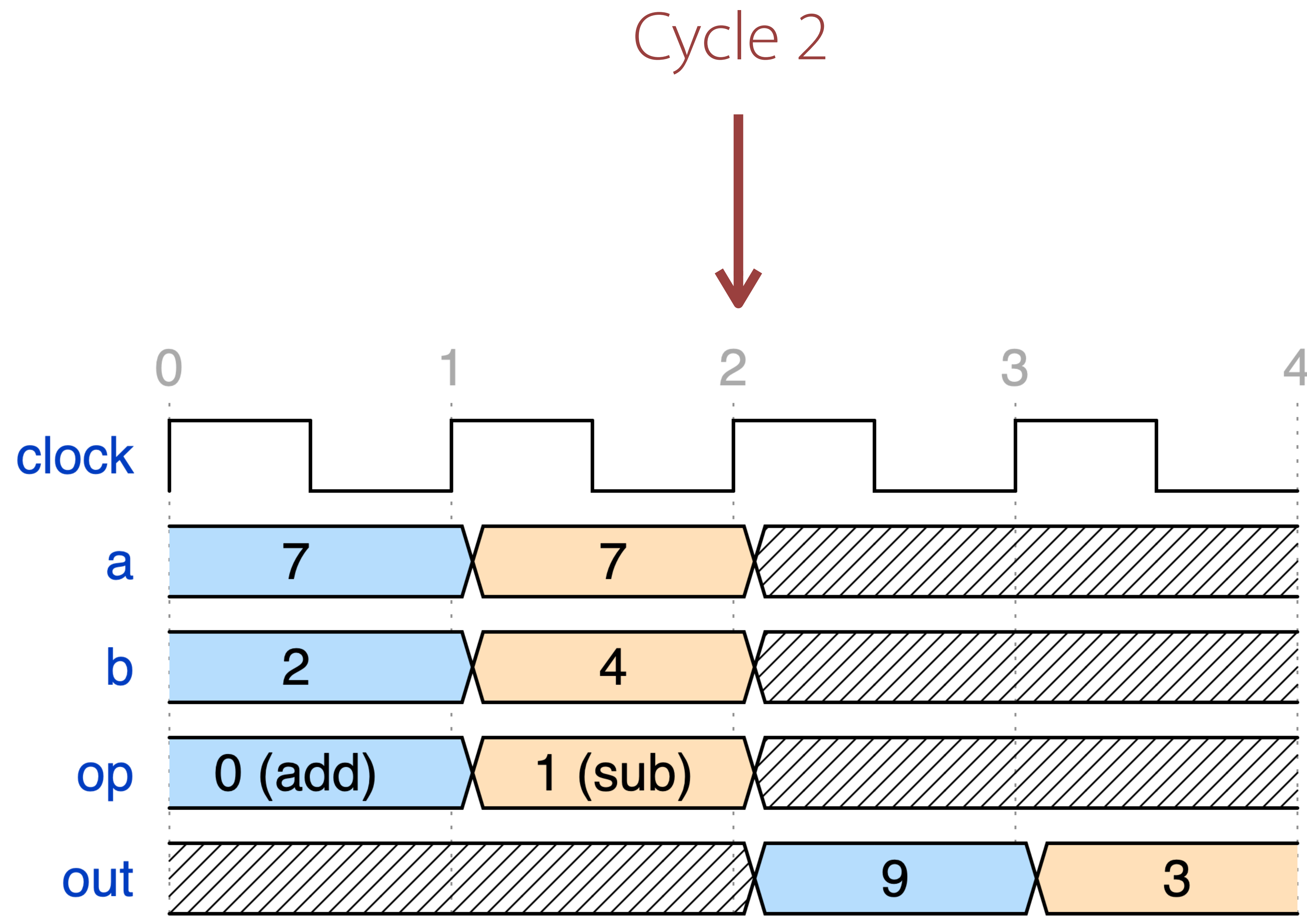
add(a=7, b=2, out=?)



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

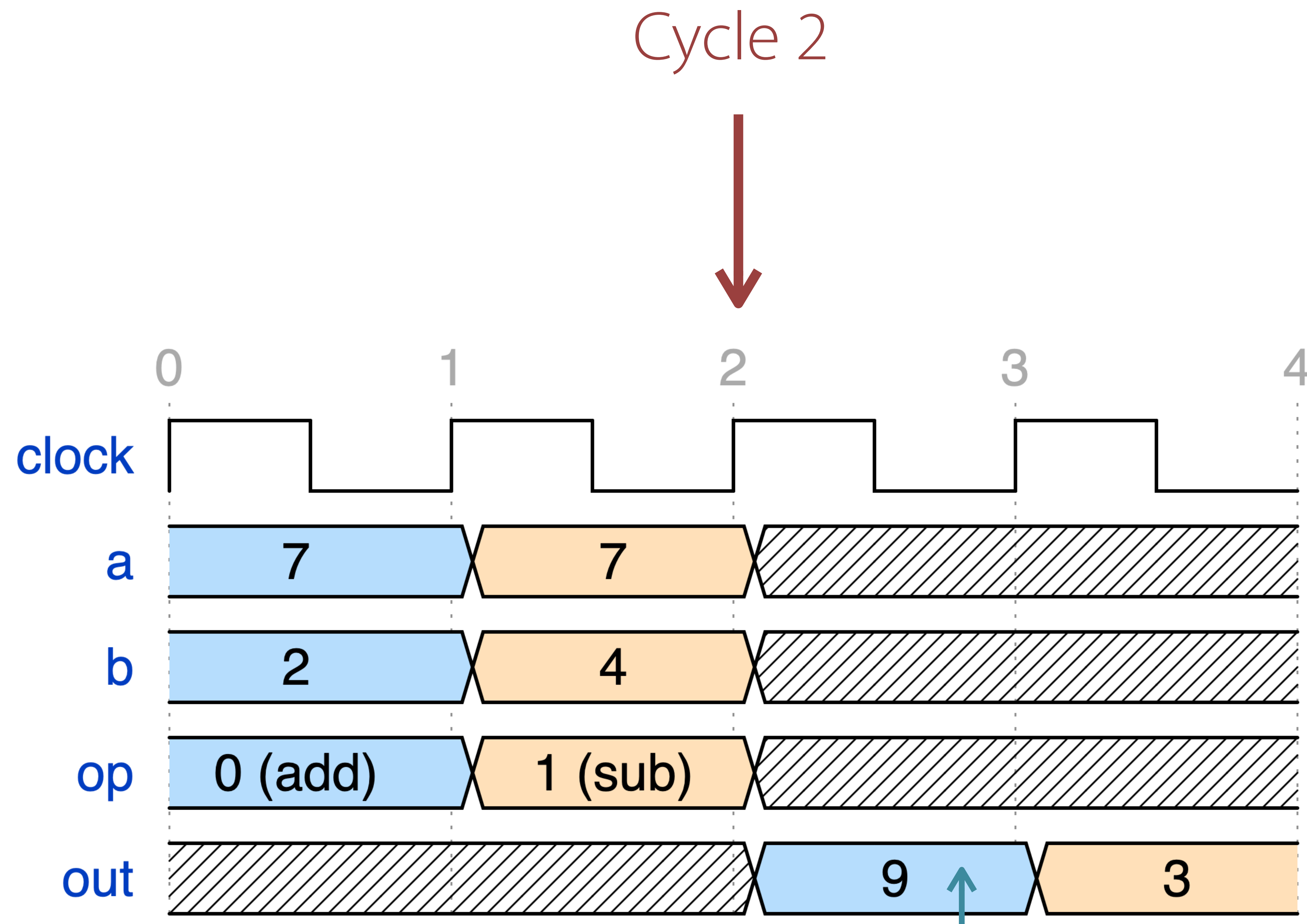
`add(a=7, b=2, out=?)`

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

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

Parent add transaction continues…

Cycle 2

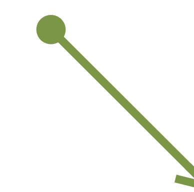


At current clock cycle: `out = 9`

```
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=?)`

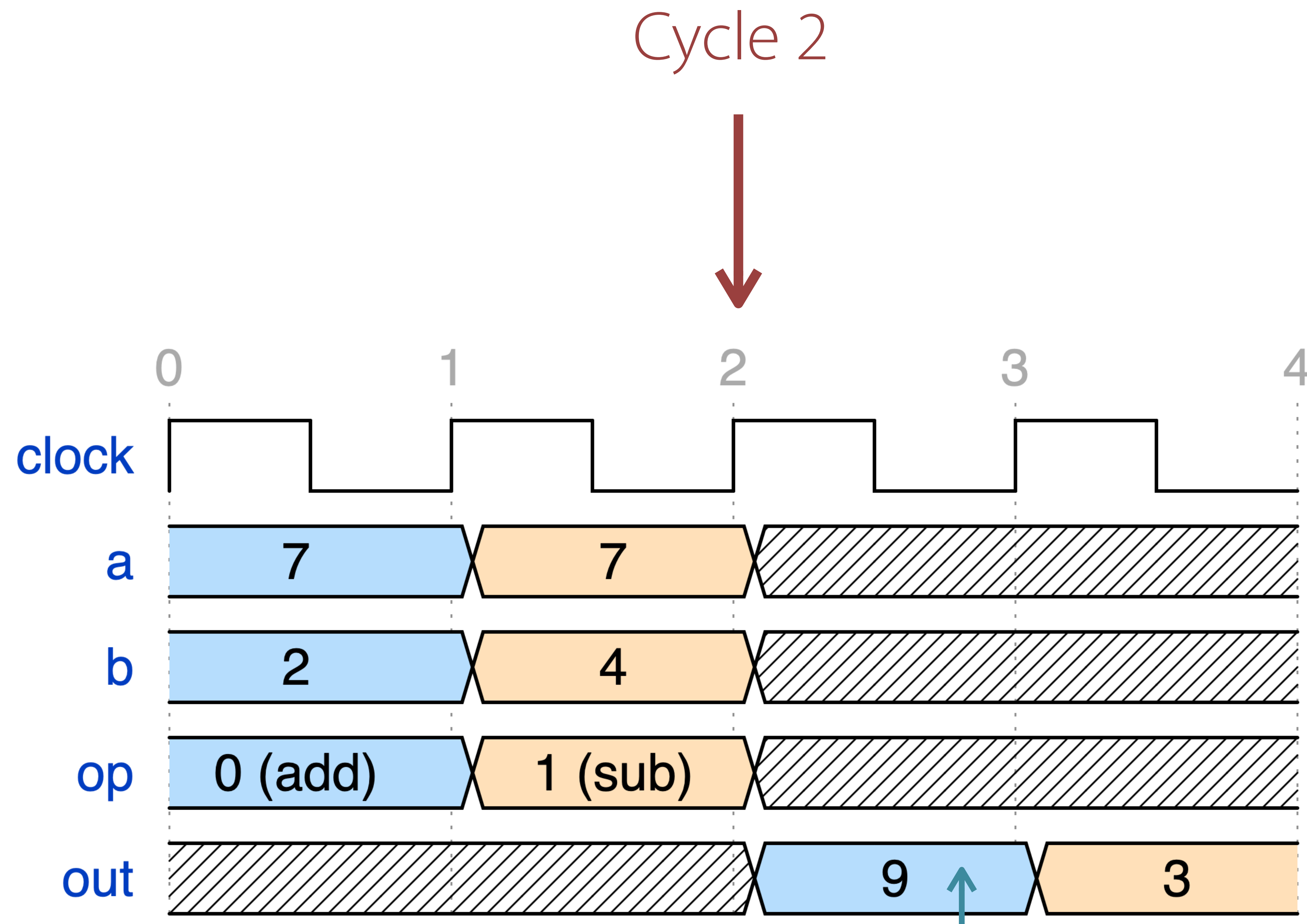


`sub(a=7, b=4, out=?)`

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

Parent add transaction continues…

Cycle 2



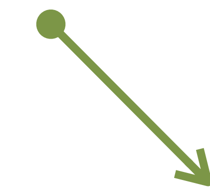
At current clock cycle: out = 9

```
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!

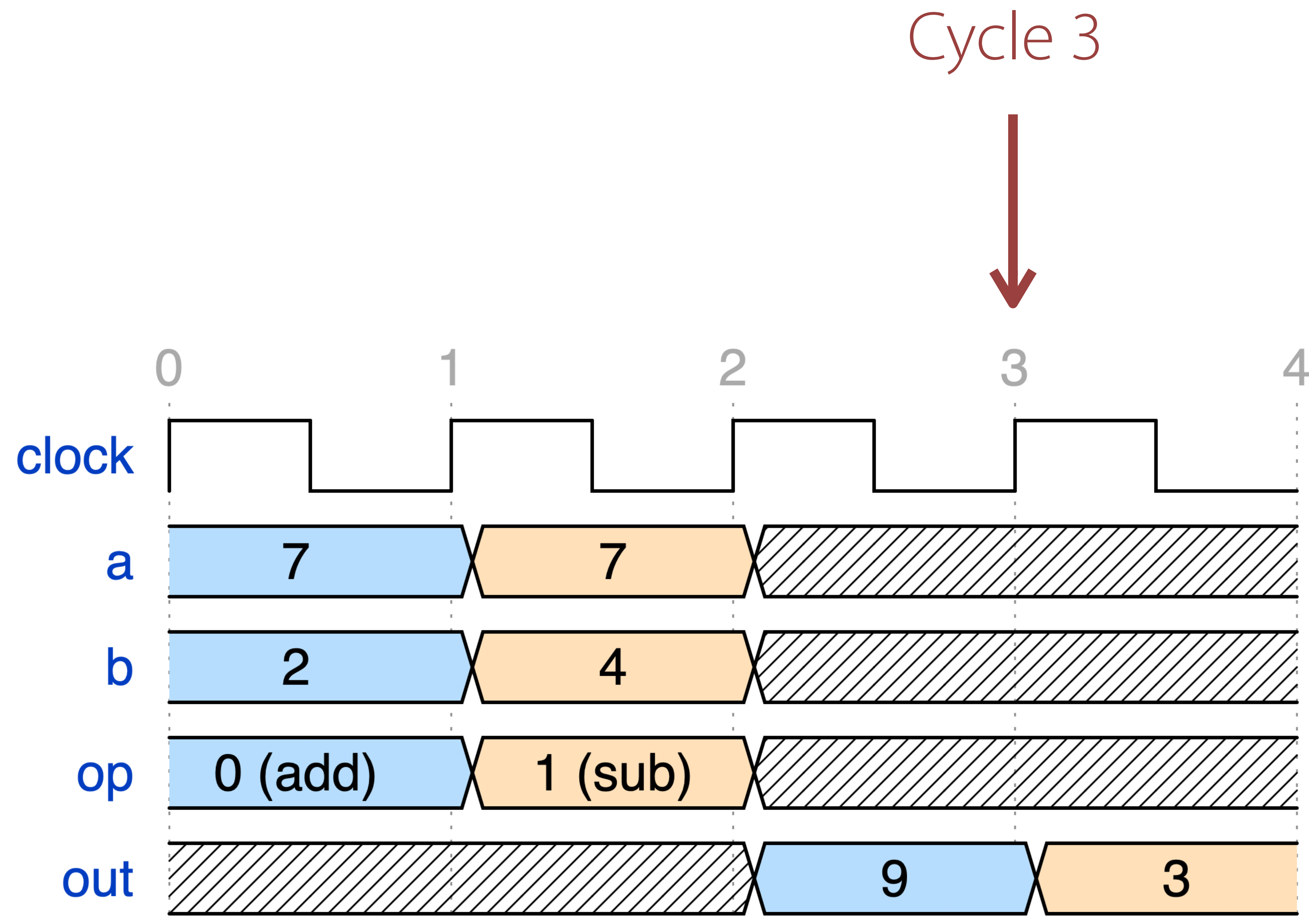
`add(a=7, b=2, out=9)`



`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);
}
```

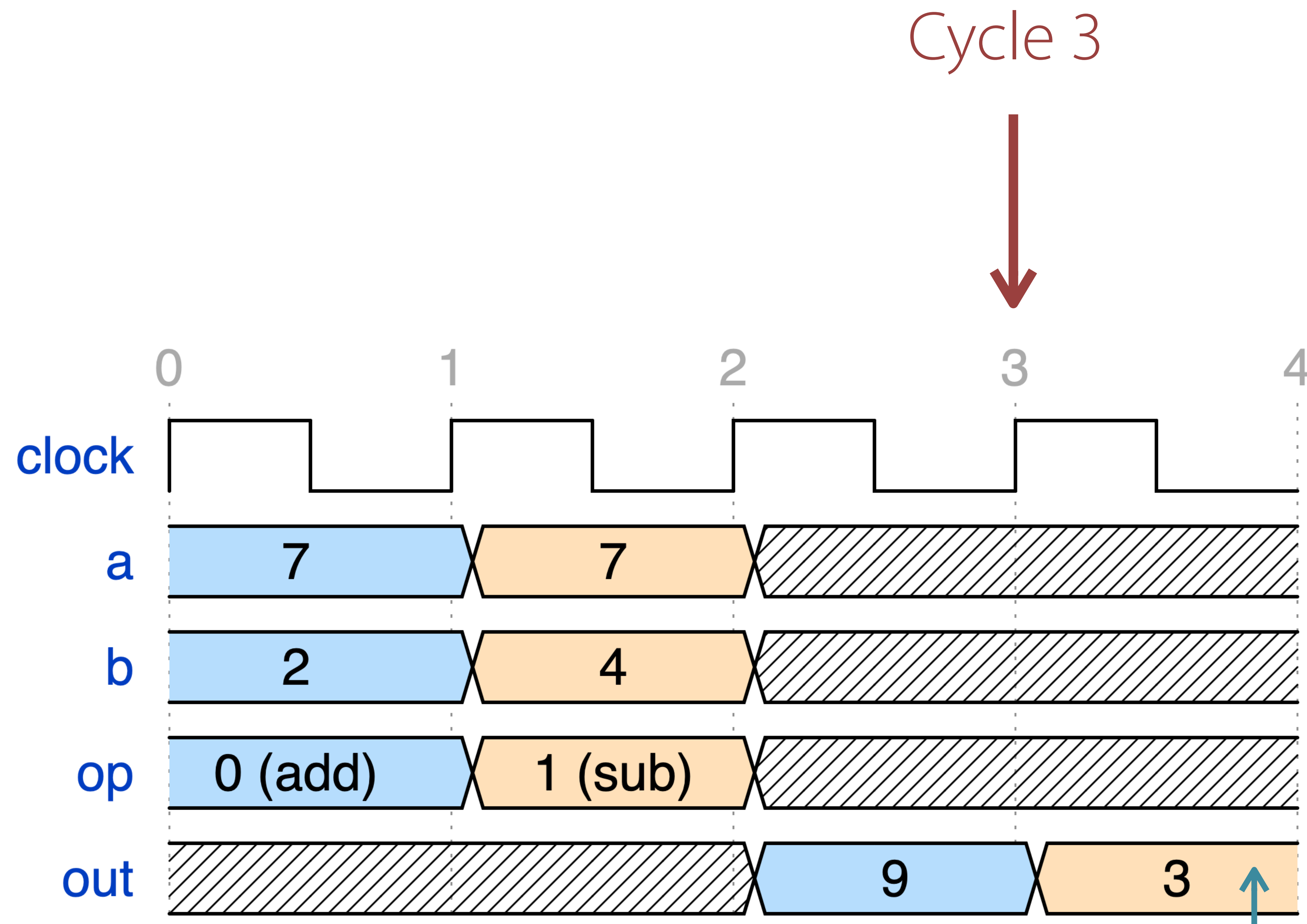
Value of out inferred from waveform

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);
}
```

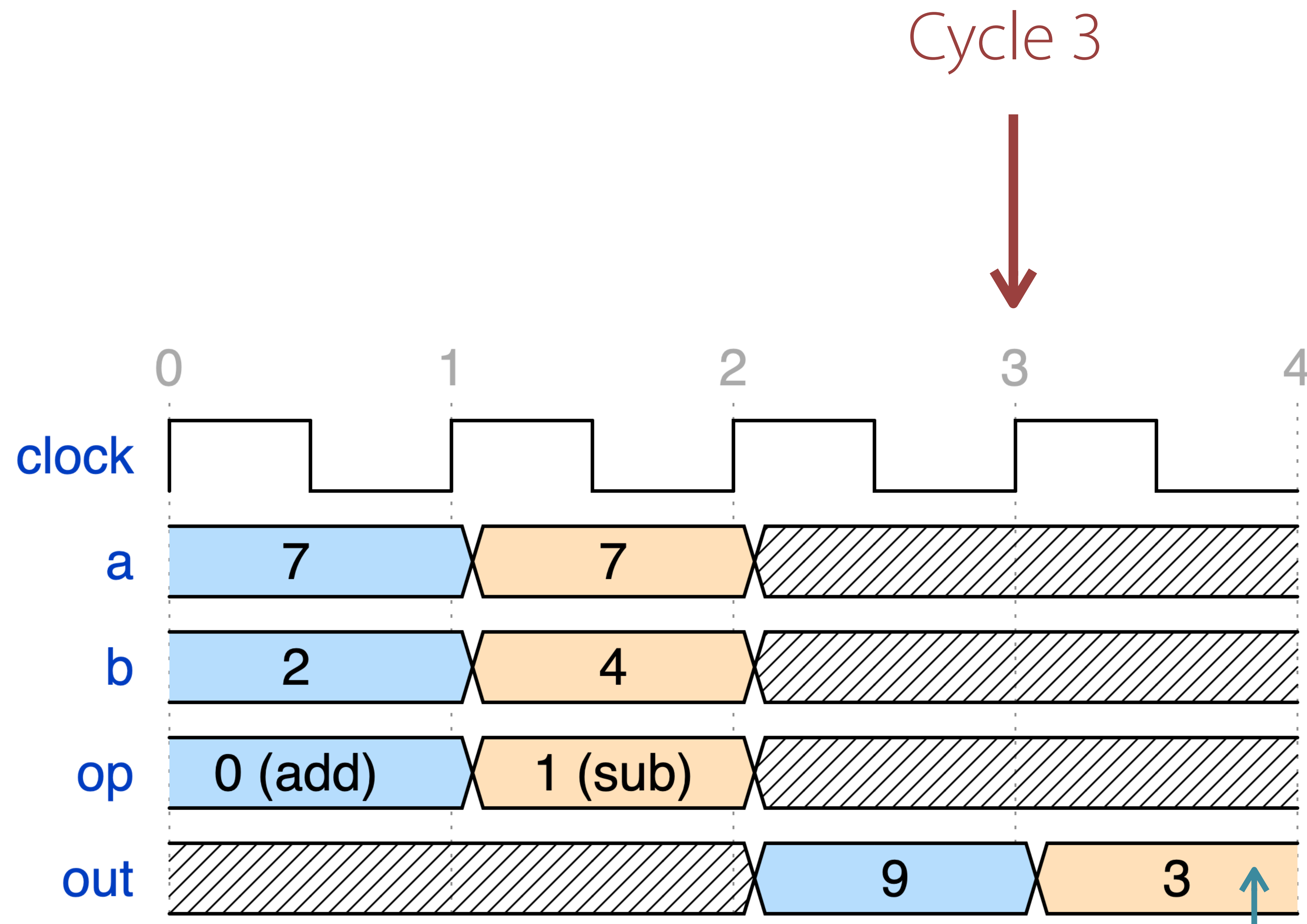
Value of `out` inferred from waveform

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!

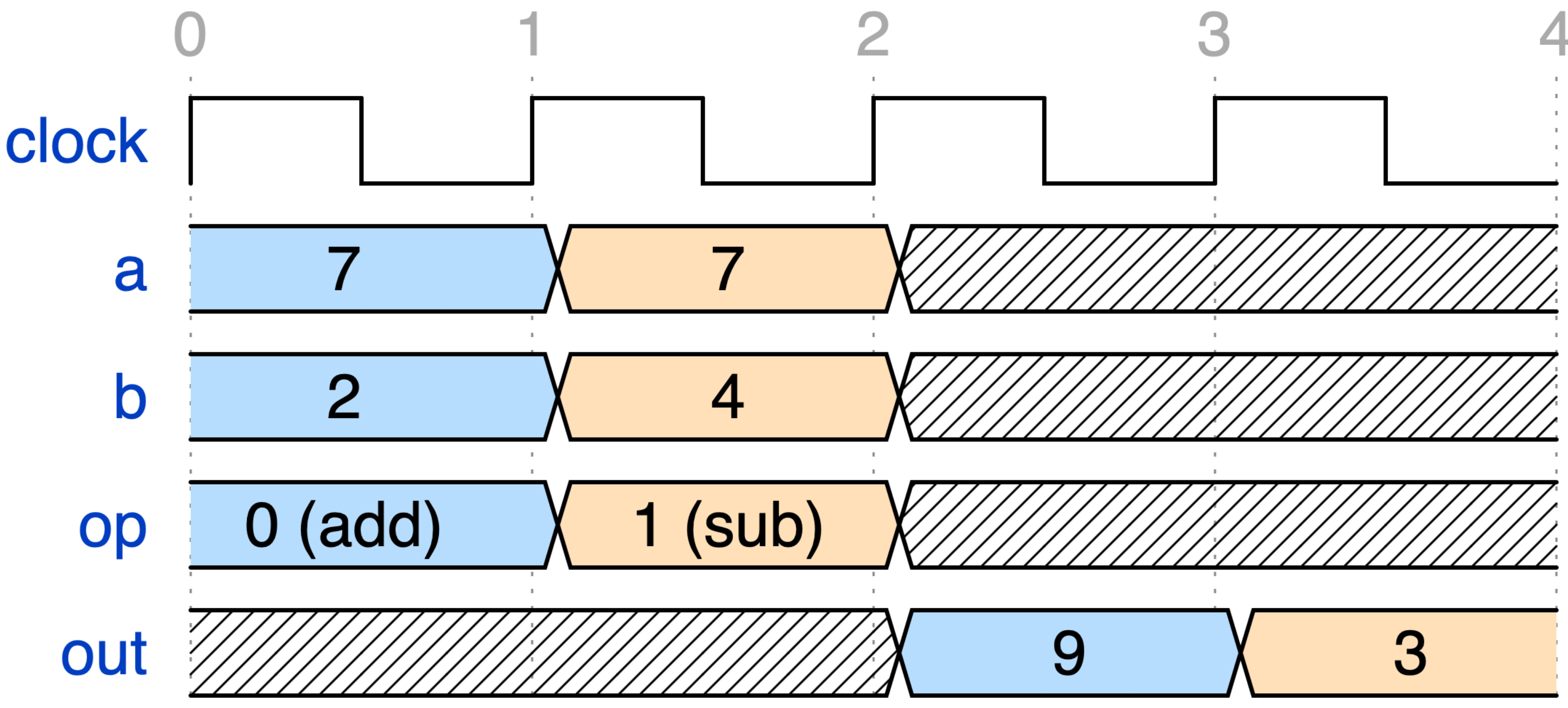
add(a=7, b=2, out=9)

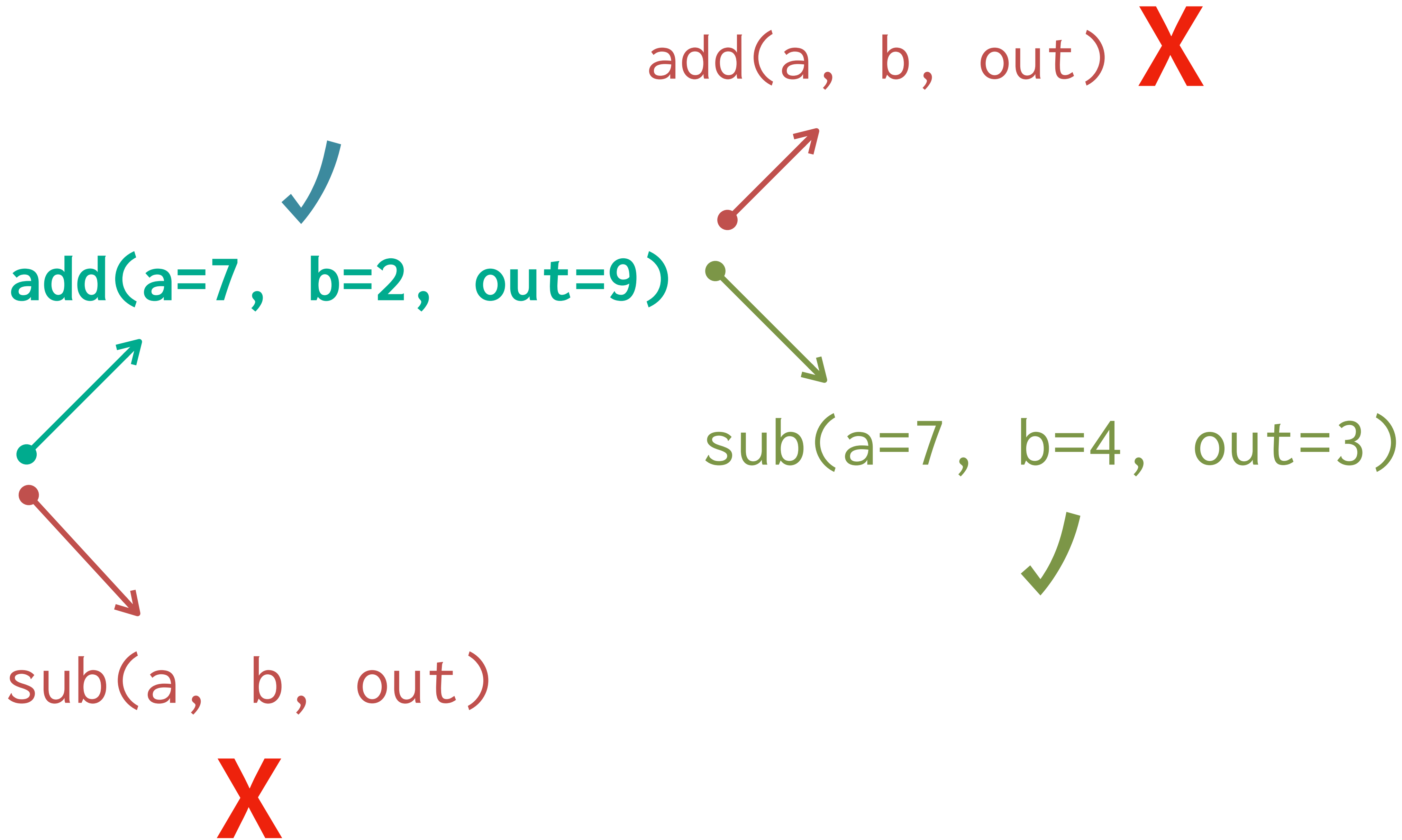
sub(a=7, b=4, out=3)

Transaction complete!

Inferred transaction trace:

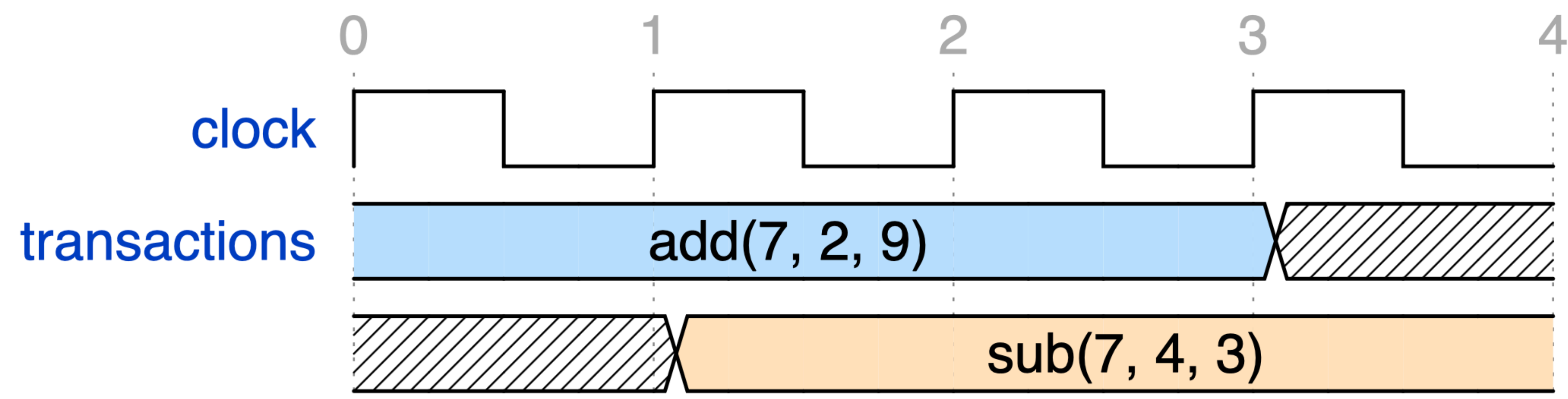
```
add(7, 2, 9); // cycles 0-3
sub(7, 4, 3); // cycles 1-4
```

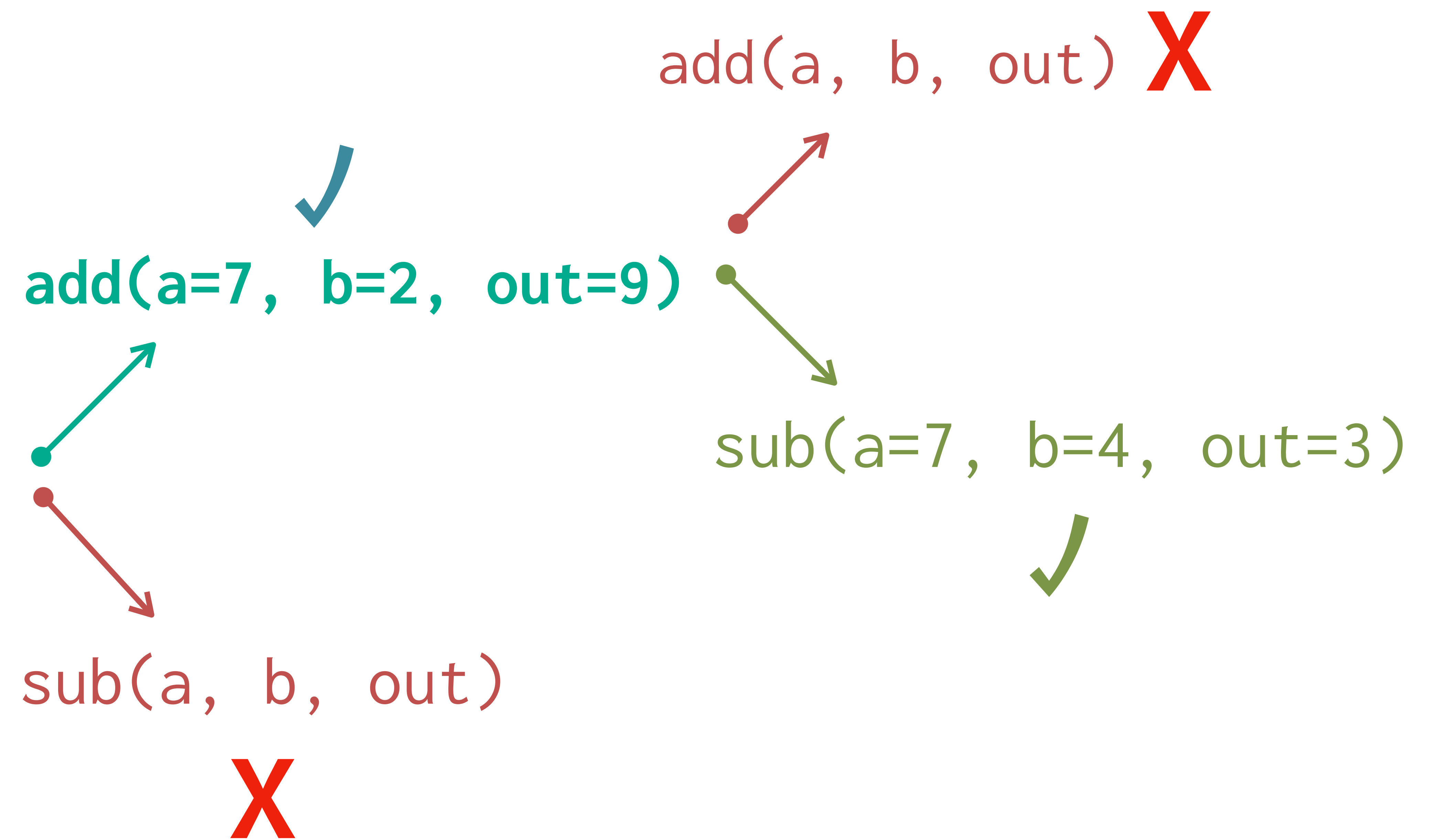




Inferred transaction trace:

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





Ongoing Evaluation

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

Ongoing Evaluation

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)

Ongoing Evaluation

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)

Wishbone

Open-source interconnect for on-chip communication

Protocol bugs in the wild

Goal: demonstrate that the reconstructor can facilitate waveform debugging

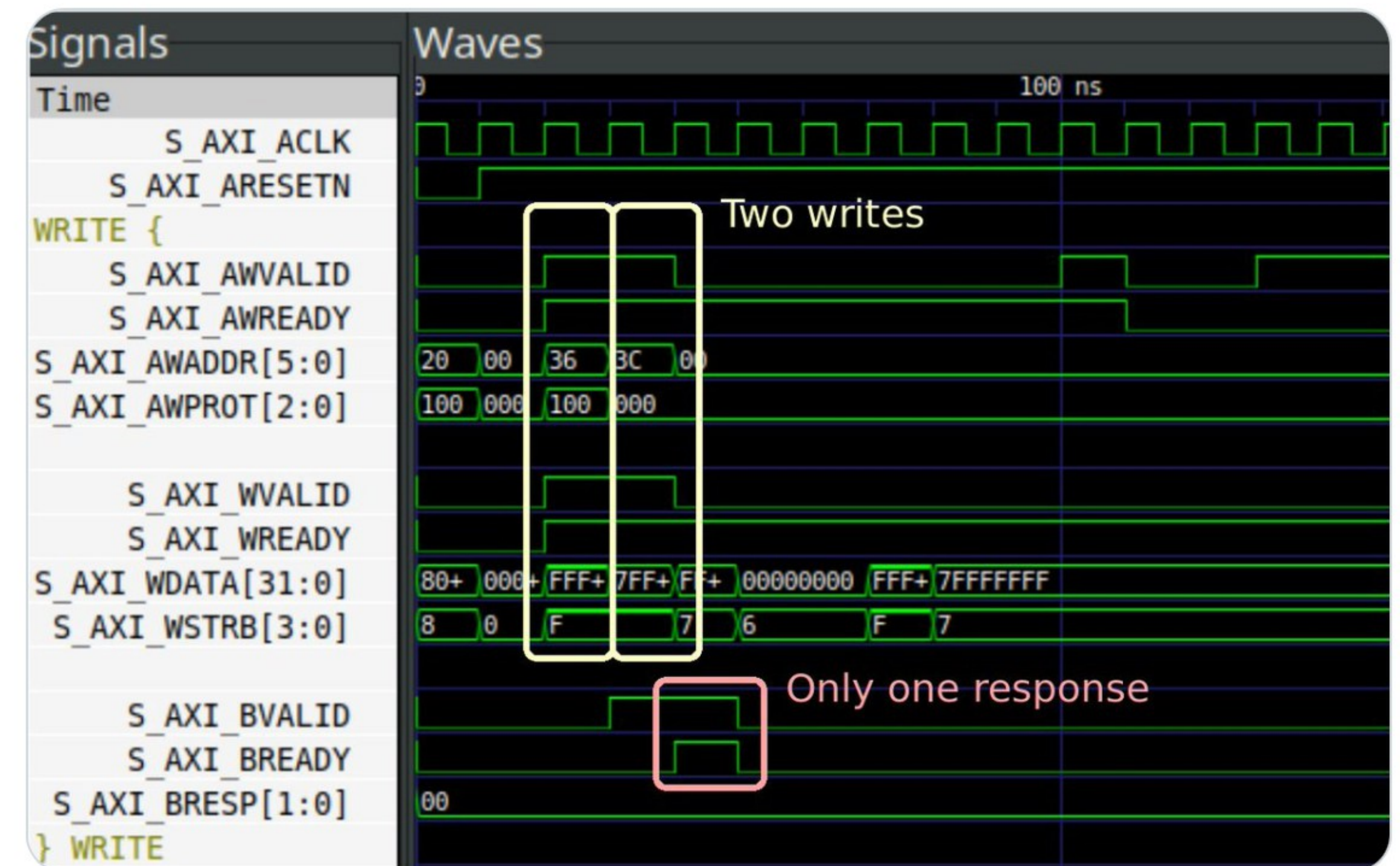
Benchmark suite of bugs in open-source FPGA designs (we focus on protocol violation bugs)

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

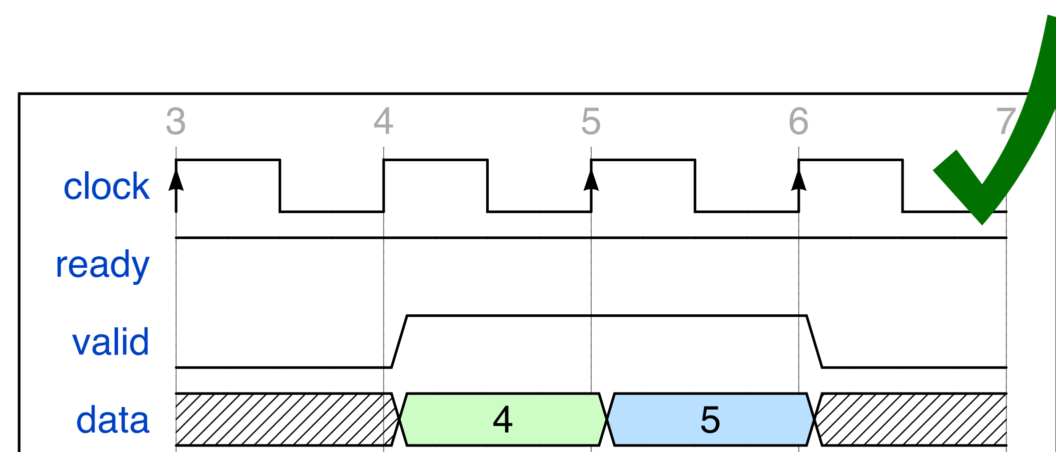
ASPLOS'22

Here's a trace from the Xilinx's "new" AXI-Lite template, as generated by Vivado 2024.2.



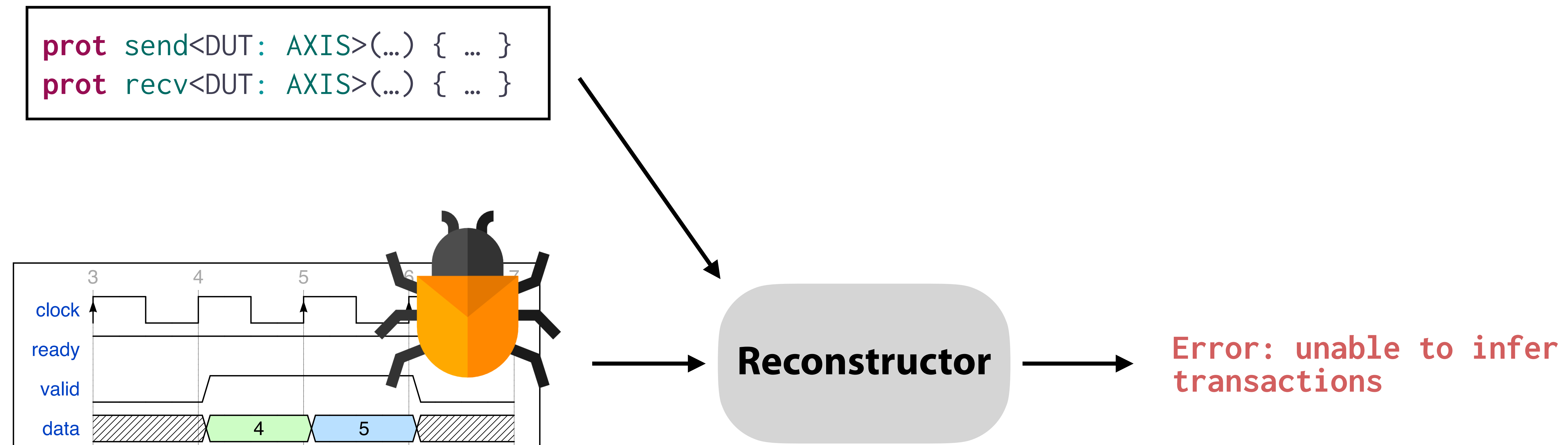
Protocol bugs in the wild

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



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

Protocol bugs in the wild

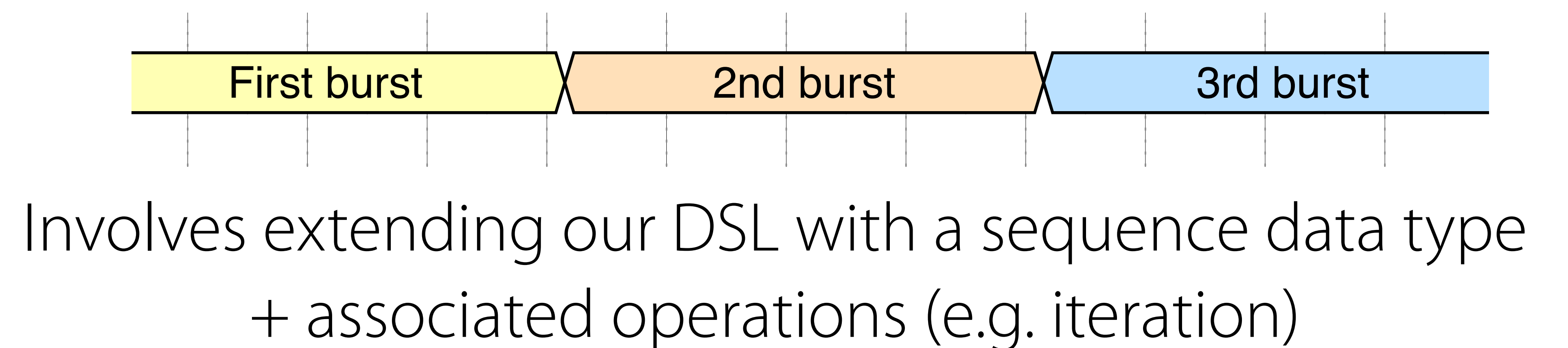


(Near-Term) Future Work

Support full AXI / AXI-Lite

Involves reasoning
about dependencies between
transactions on different channels

Support Wishbone's Burst Mode



Involves extending our DSL with a sequence data type
+ associated operations (e.g. iteration)

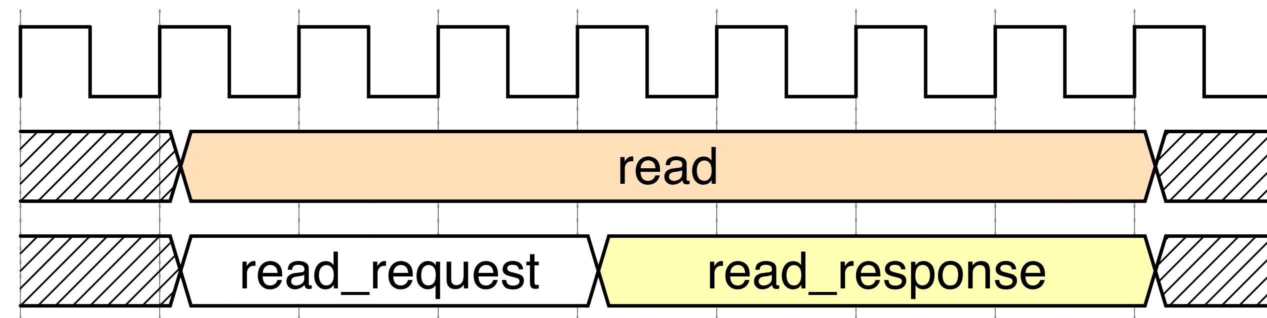
Future work: *Transaction-level* visualizations

Future work: *Transaction-level* visualizations

1. Augmented waveform viewer

Visualize inferred transactions in the Surfer waveform viewer

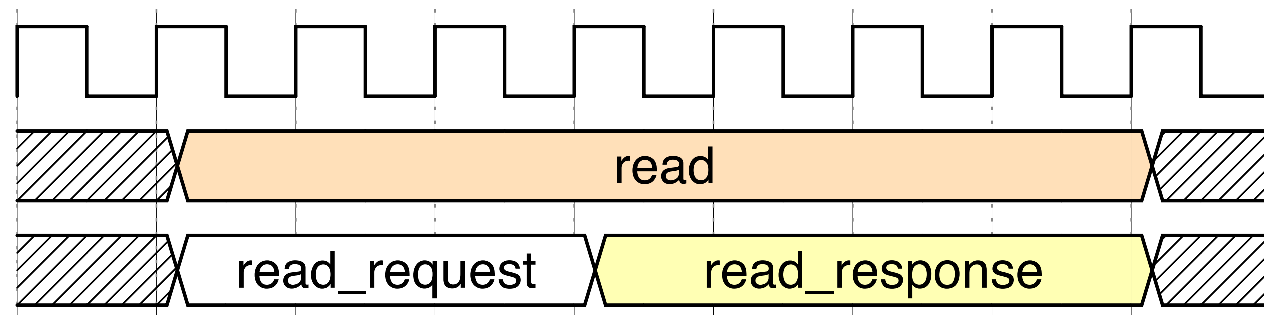
[Skarman et al. CAV '25]



Future work: *Transaction-level* visualizations

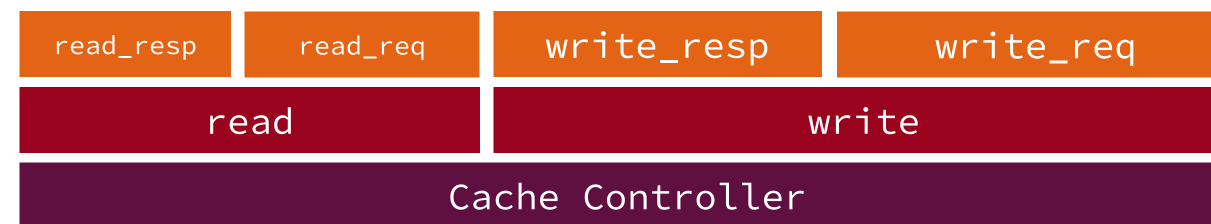
1. Augmented waveform viewer

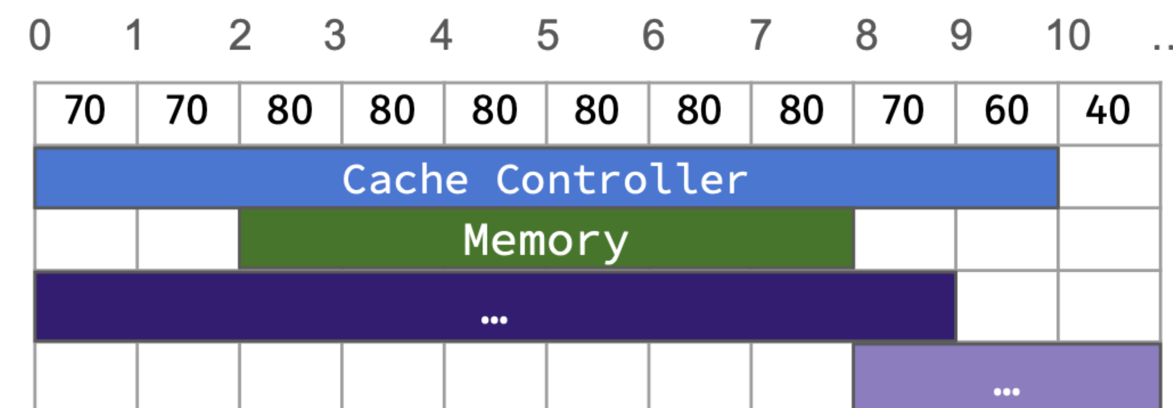
Visualize inferred transactions in the Surfer waveform viewer [Skarman et al. CAV '25]



2. Transaction profiler

Visualizes performance bottlenecks

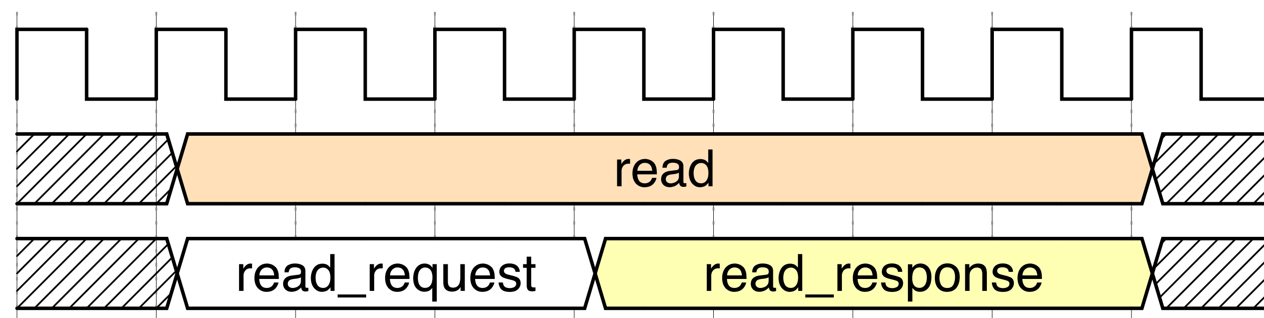




Future work: *Transaction-level* visualizations

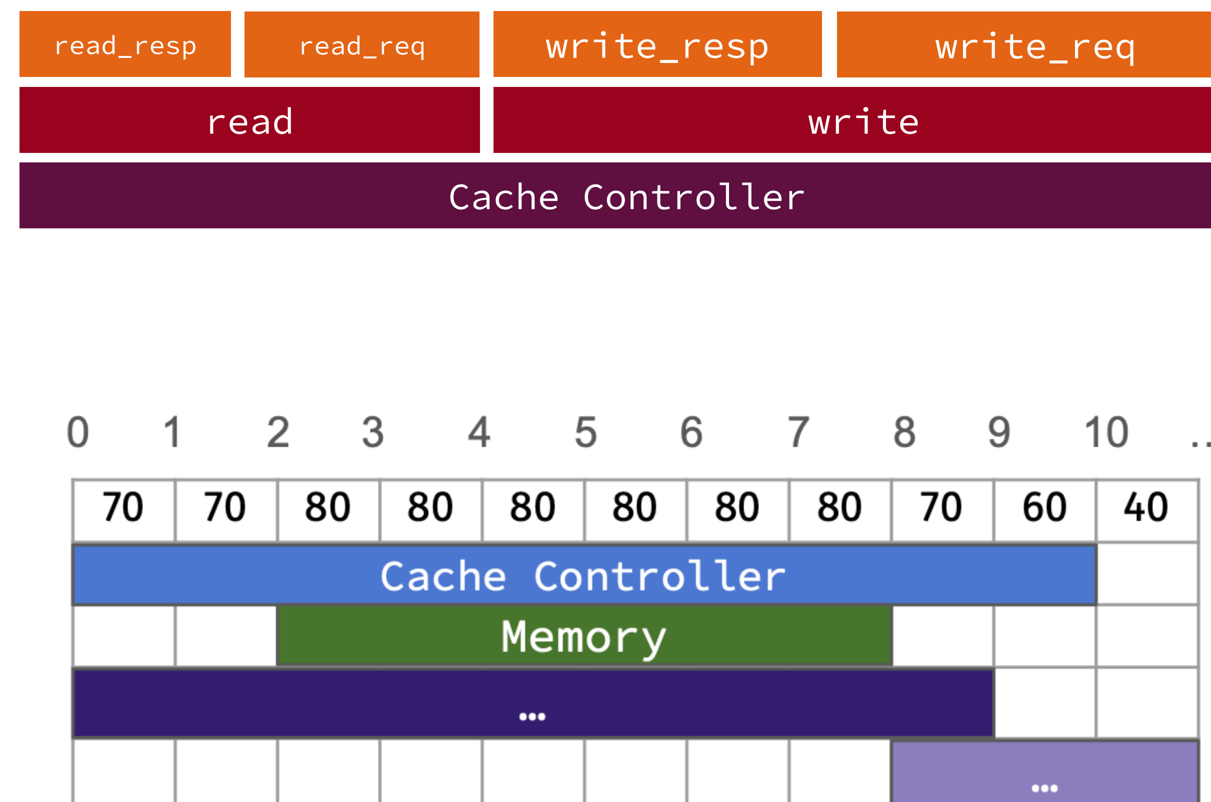
1. Augmented waveform viewer

Visualize inferred transactions in the Surfer waveform viewer [Skarman et al. CAV '25]



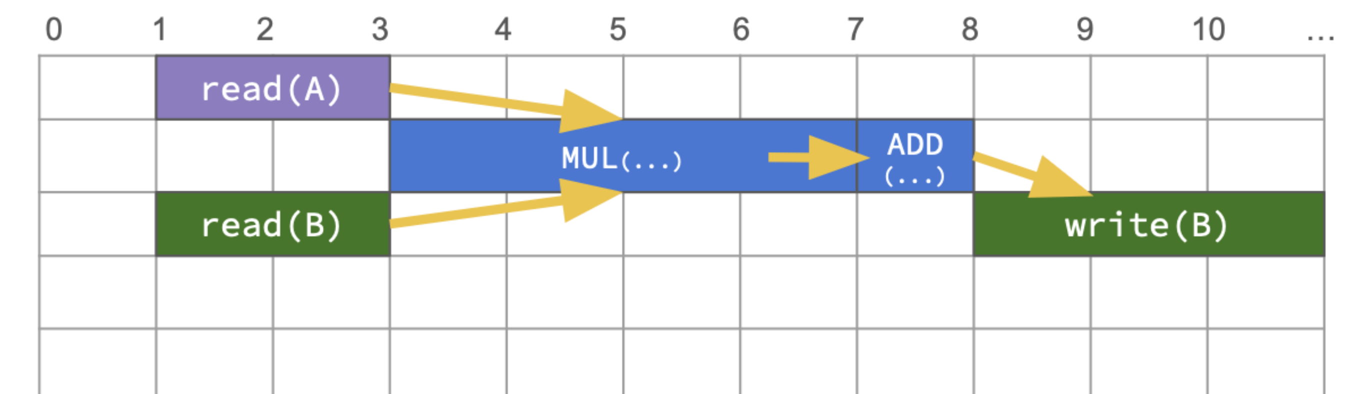
2. Transaction profiler

Visualizes performance bottlenecks



3. Dependency tracker

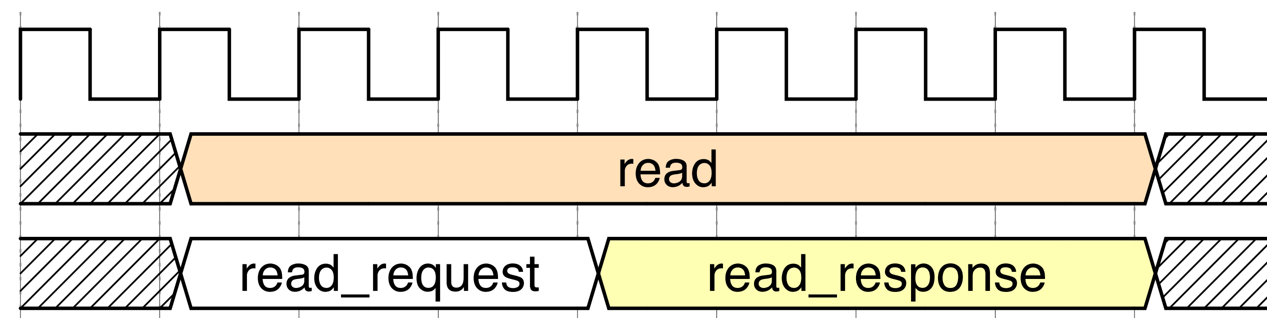
Visualizes & monitors inter-transaction dependencies



Future work: *Transaction-level* visualizations

1. Augmented waveform viewer

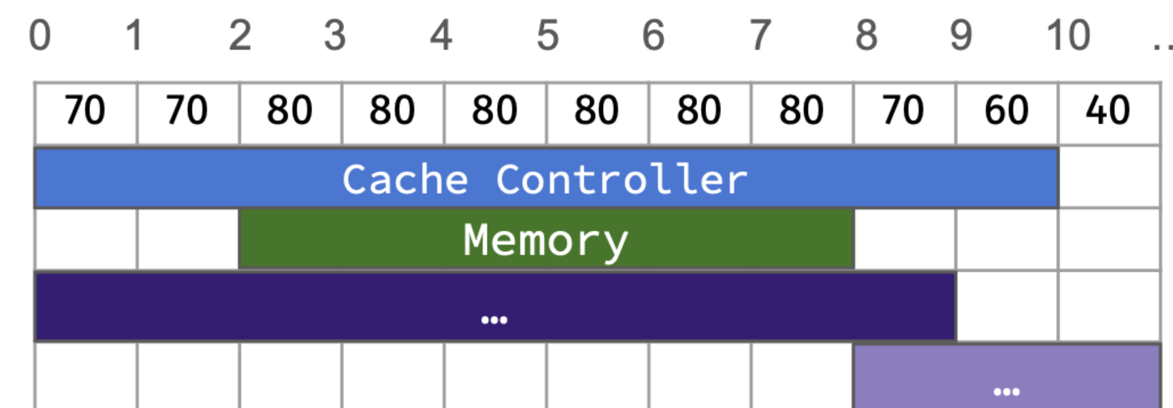
Visualize inferred transactions in the Surfer waveform viewer [Skarman et al. CAV '25]



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

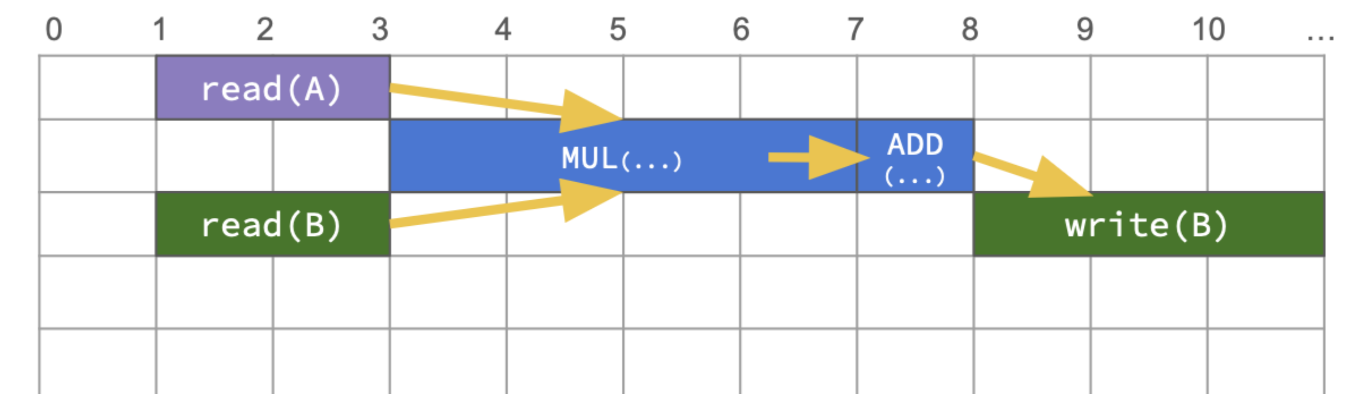
2. Transaction profiler

Visualizes performance bottlenecks



3. Dependency tracker

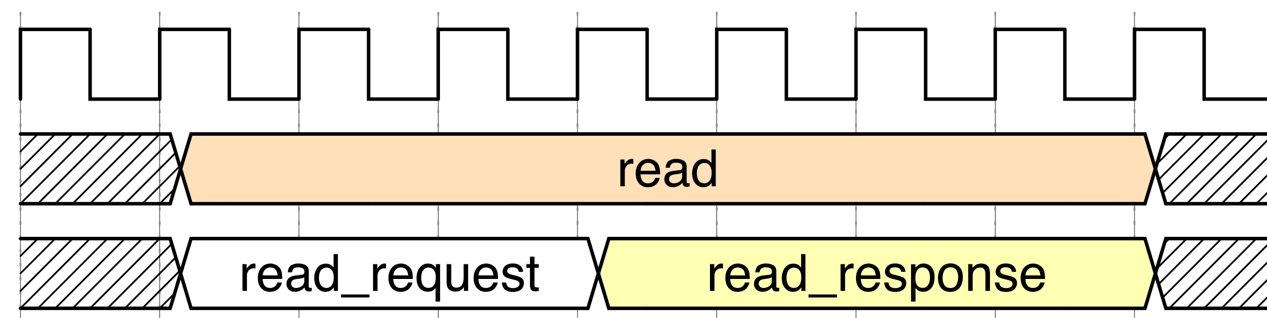
Visualizes & monitors inter-transaction dependencies



Future work: *Transaction-level* visualizations

1. Augmented waveform viewer

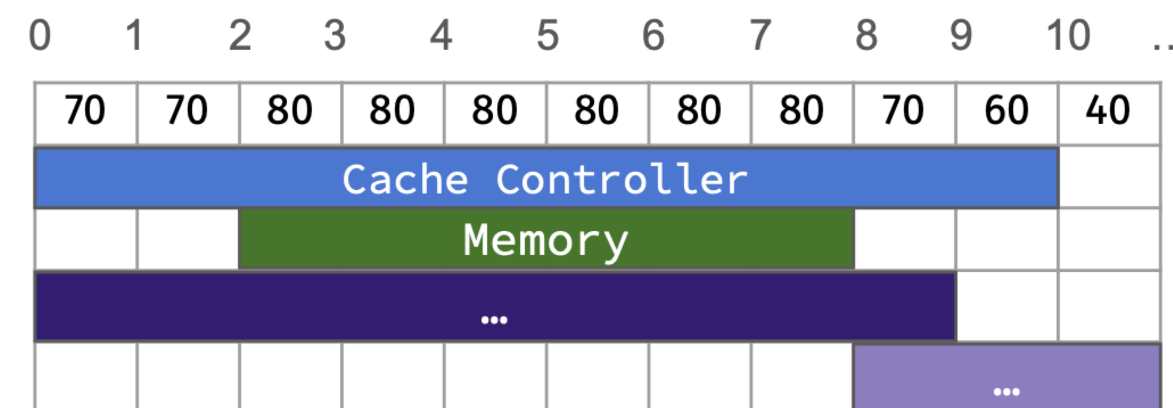
Visualize inferred transactions in the Surfer waveform viewer [Skarman et al. CAV '25]



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

2. Transaction profiler

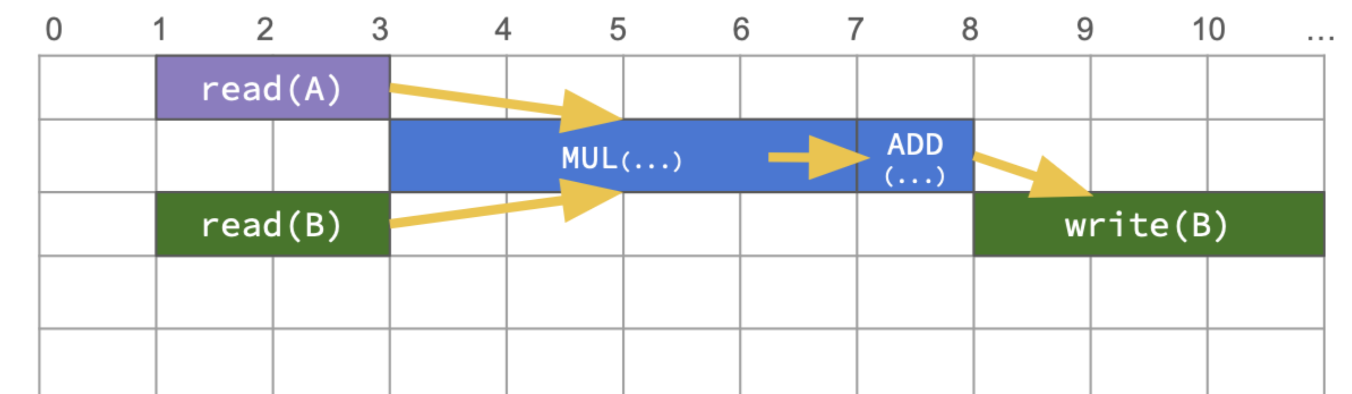
Visualizes performance bottlenecks



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

3. Dependency tracker

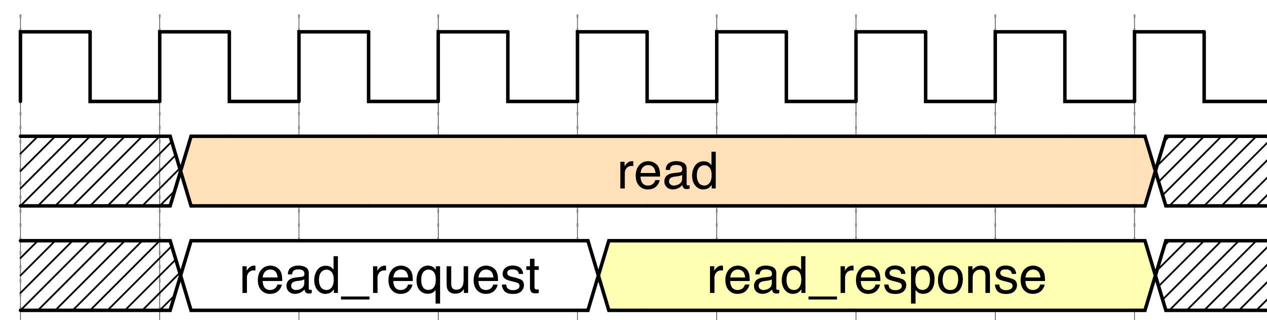
Visualizes & monitors inter-transaction dependencies



Future work: *Transaction-level* visualizations

1. Augmented waveform viewer

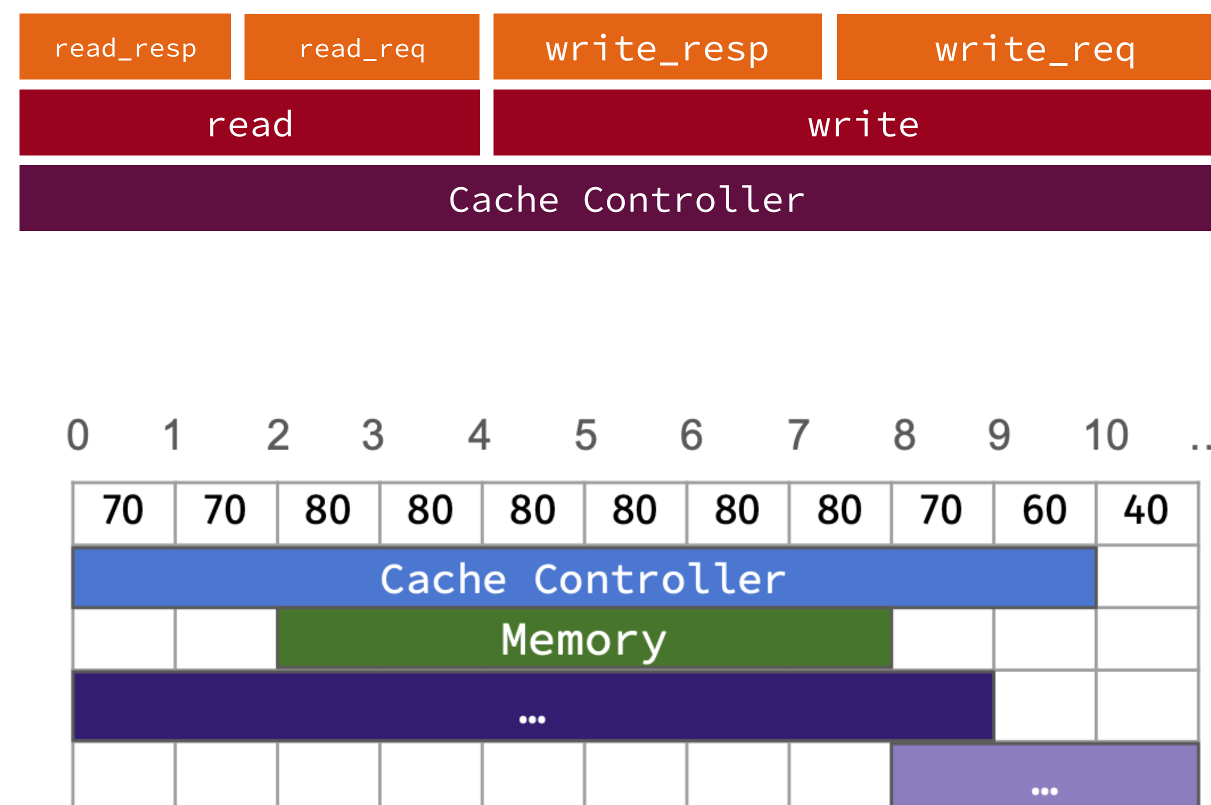
Visualize inferred transactions in the Surfer waveform viewer [Skarman et al. CAV '25]



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

2. Transaction profiler

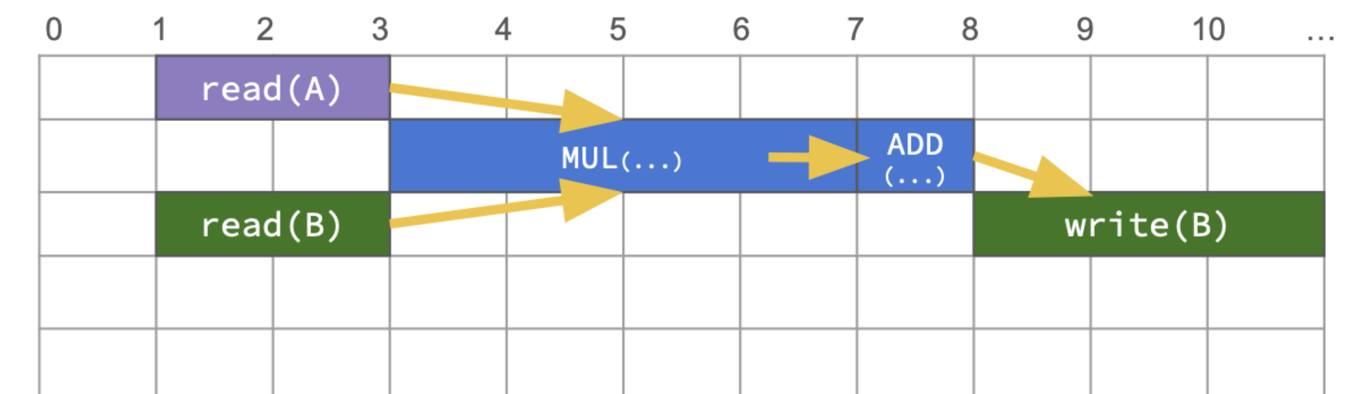
Visualizes performance bottlenecks



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

3. Dependency tracker

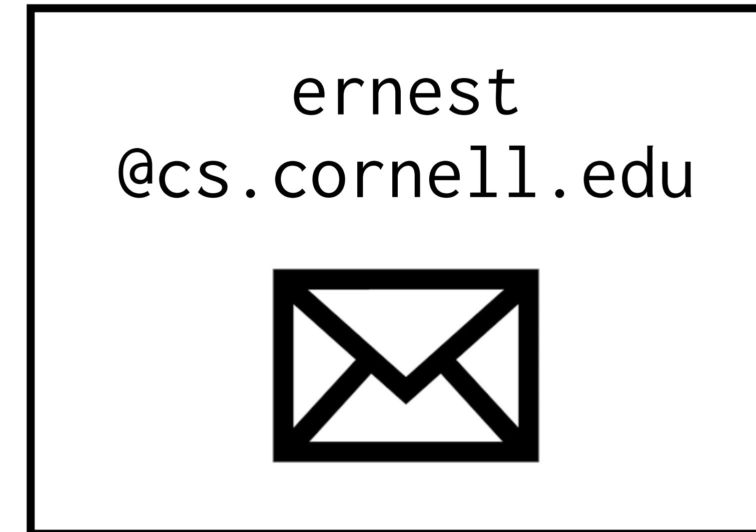
Visualizes & monitors inter-transaction dependencies

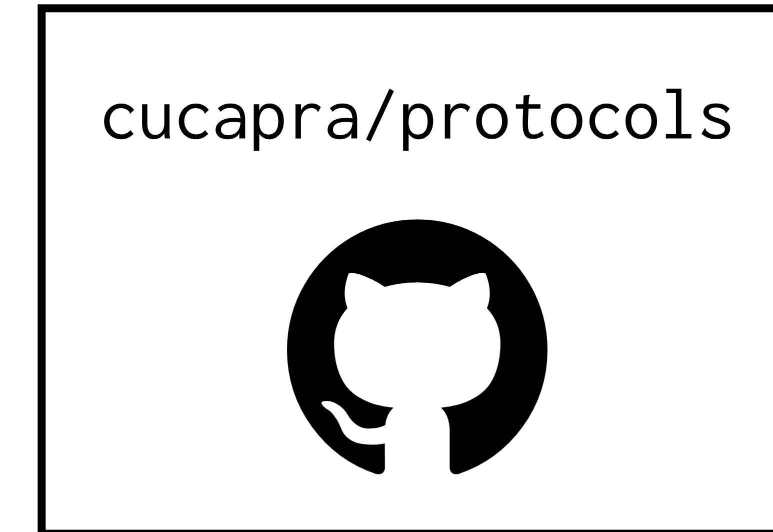


Assists engineers with dependency tracking at a higher abstraction level

Conclusion: Hardware communication protocols can be specified as programs!

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.

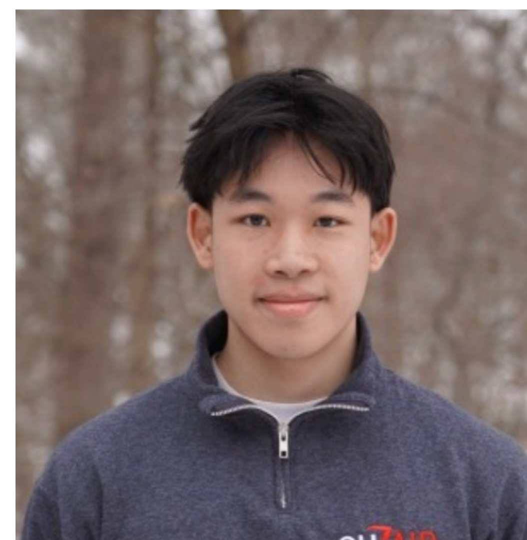




Conclusion: Hardware communication protocols can be specified as programs!
Using our DSL, the same protocol spec can be used to both run tests & infer transactions.

Ernest
Ng

Nikil
Shyamsunder

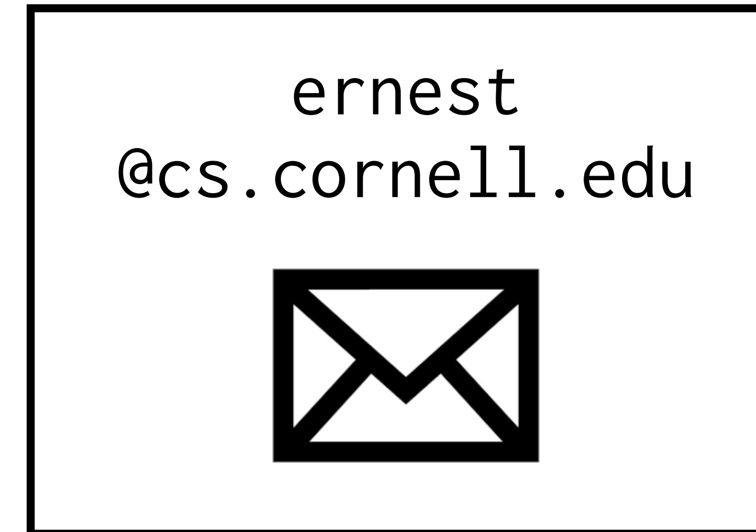


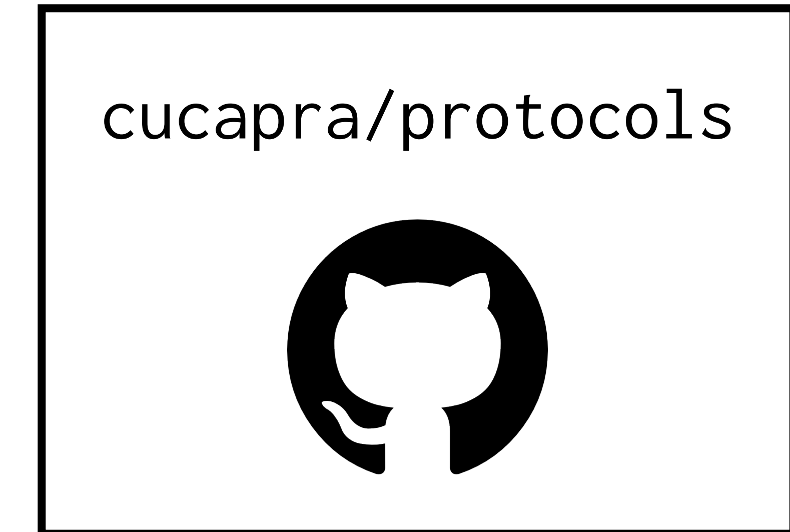
Francis
Pham

Adrian
Sampson

Kevin
Laeufer

Thanks!





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

Appendix

Other Future Work

Other Future Work

Sub-protocols

```
prot transfer(addr: u32, data: u32) { … }

prot block_transfer(addr: u32, data: [u32]) {
   for item in data {
      transfer(addr, data);
      step();
   }
}
```

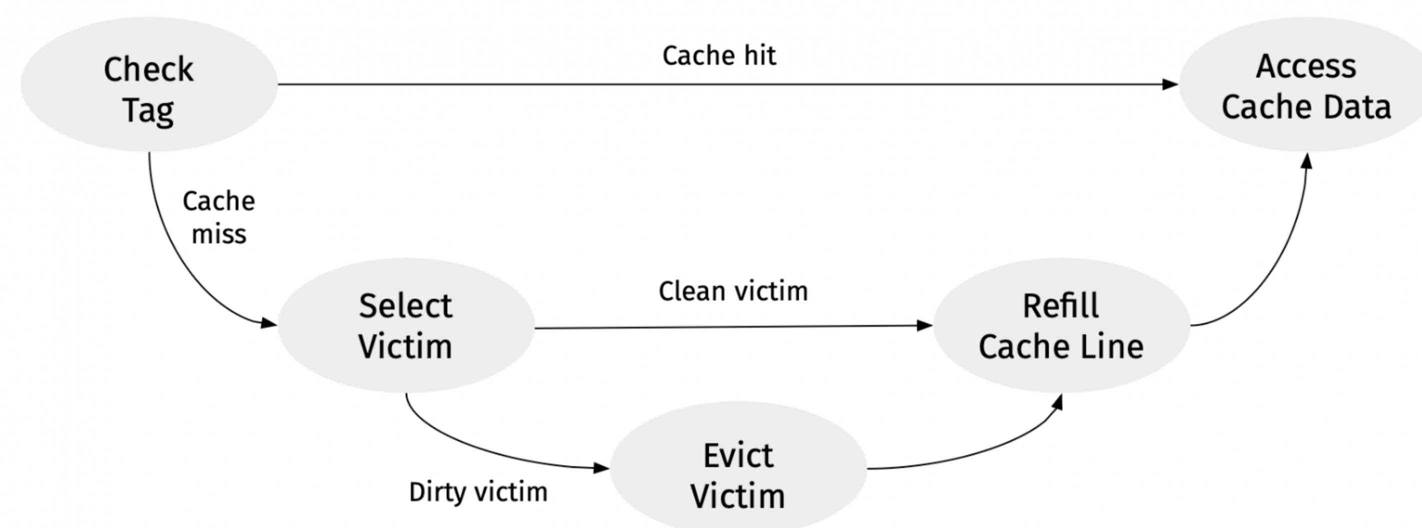
Other Future Work

Sub-protocols

```
prot transfer(addr: u32, data: u32) { … }

prot block_transfer(addr: u32, data: [u32]) {
   for item in data {
      transfer(addr, data);
      step();
   }
}
```

In-hardware monitoring

Compile protocols to an automata-based IR, and ultimately FSMs in Verilog



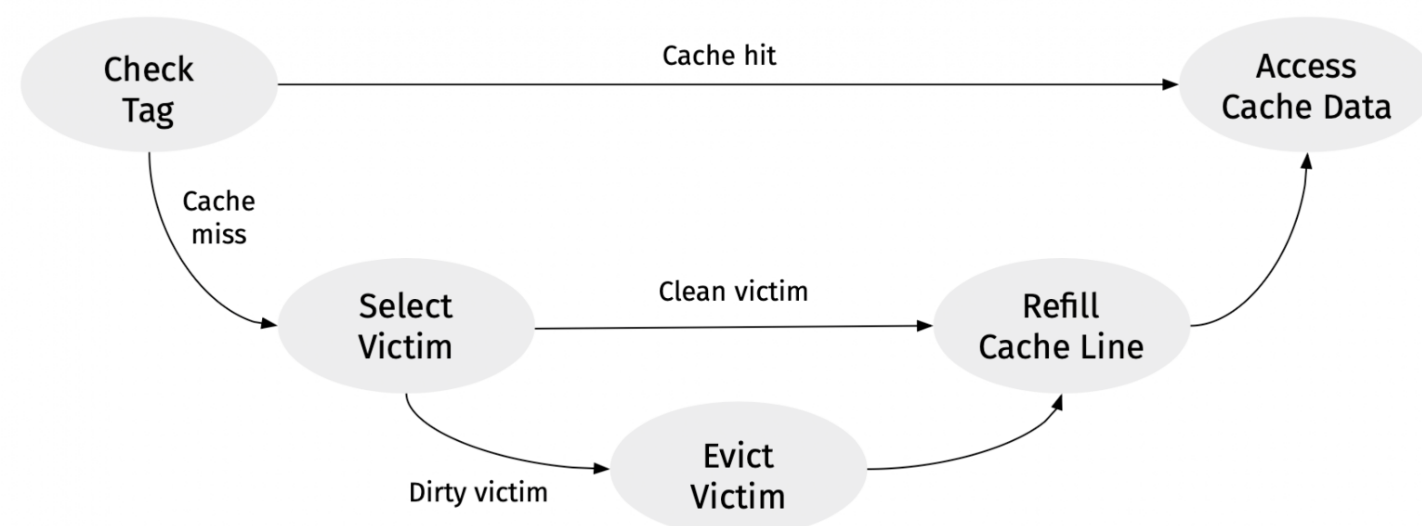
Other Future Work

Sub-protocols

```
prot transfer(addr: u32, data: u32) { … }

prot block_transfer(addr: u32, data: [u32]) {
   for item in data {
      transfer(addr, data);
      step();
   }
}
```

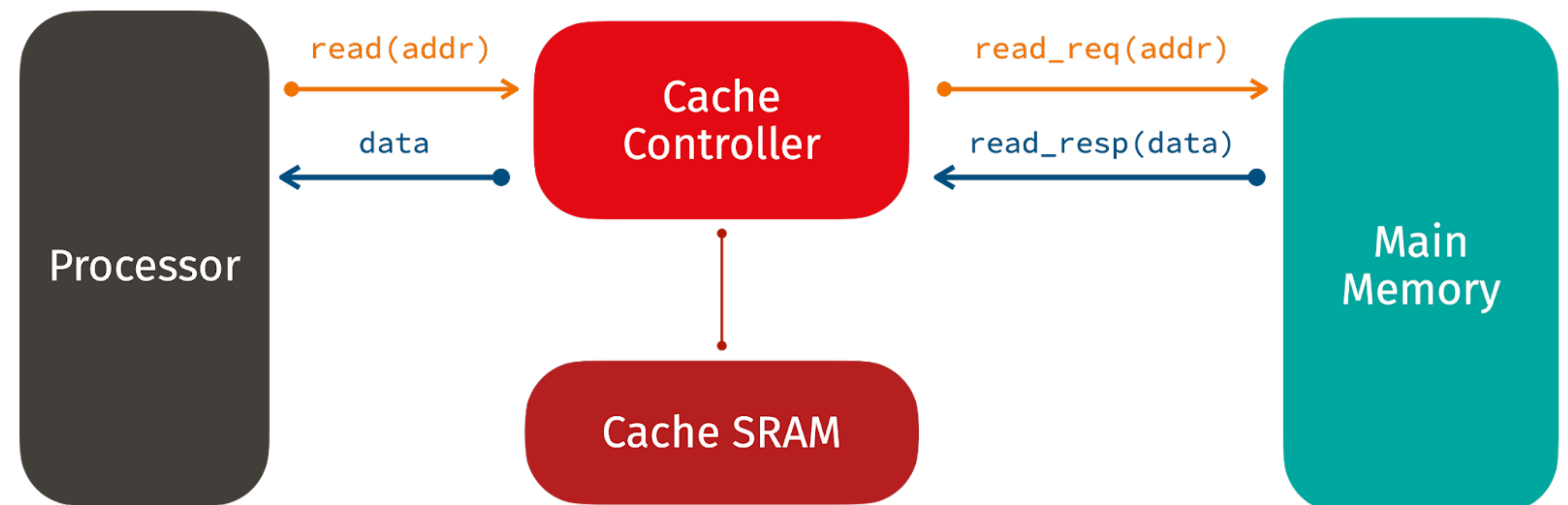
In-hardware monitoring

Compile protocols to an automata-based IR, and ultimately FSMs in Verilog



Support higher-level protocols

Model protocols that involve multiple devices (e.g. CXL, cache coherence)



The abstraction gap in hardware

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

The abstraction gap in hardware

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

Software

```
fn add(a: u8, b: u8) → u8;
```

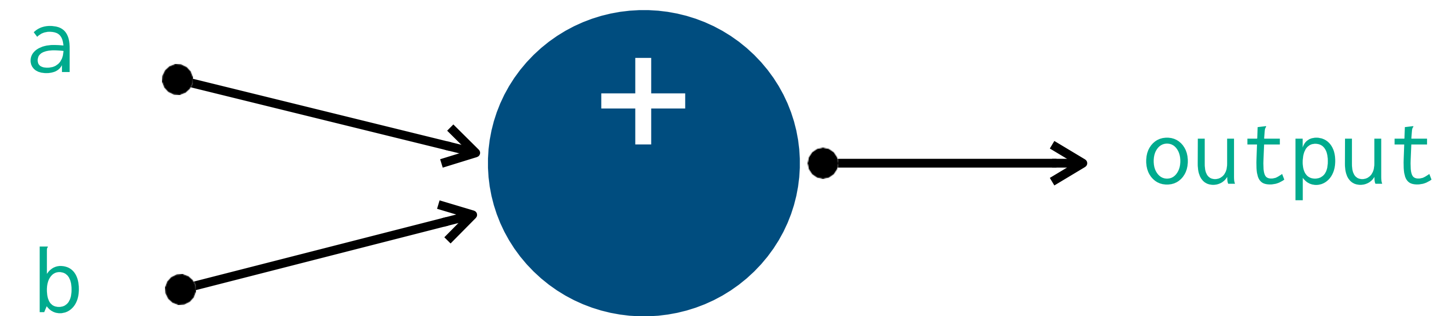
The abstraction gap in hardware

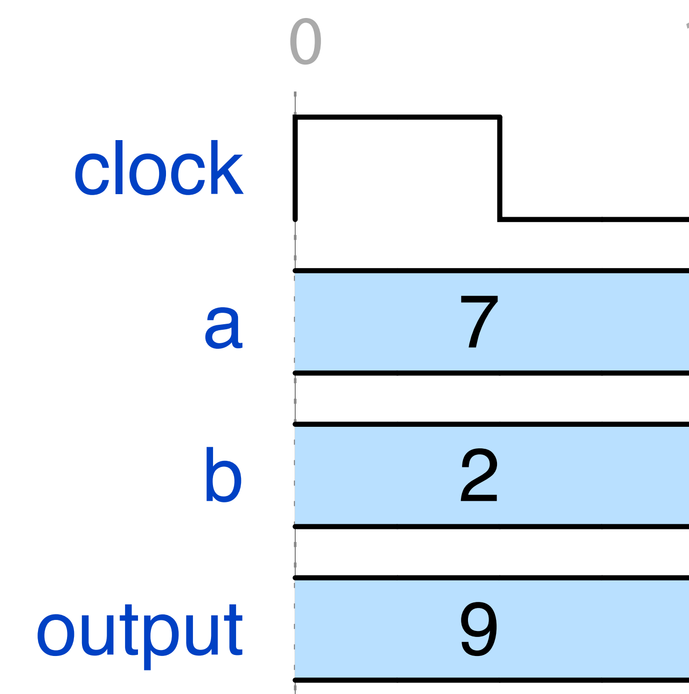
Hardware runs at the level of individual clock cycles & signals,
but we care about high-level **transactions**!

Software

```
fn add(a: u8, b: u8) → u8;
```

Combinational Adder





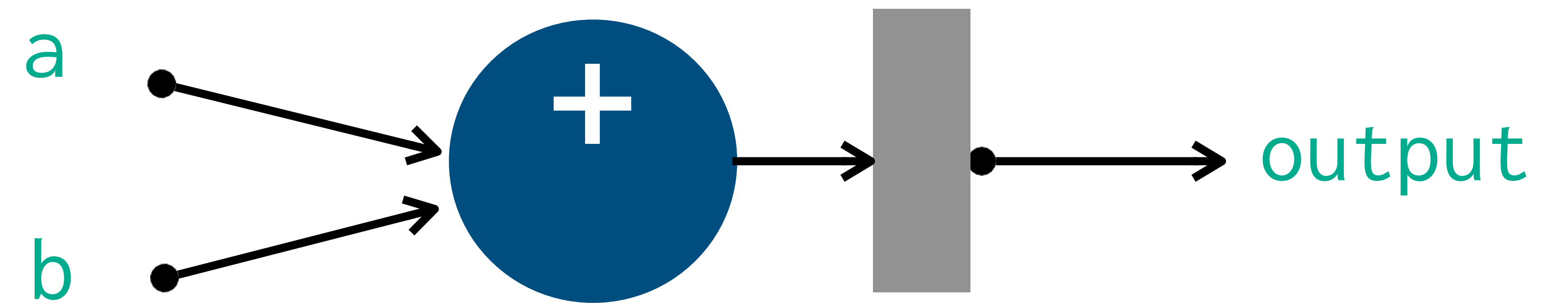
The abstraction gap in hardware

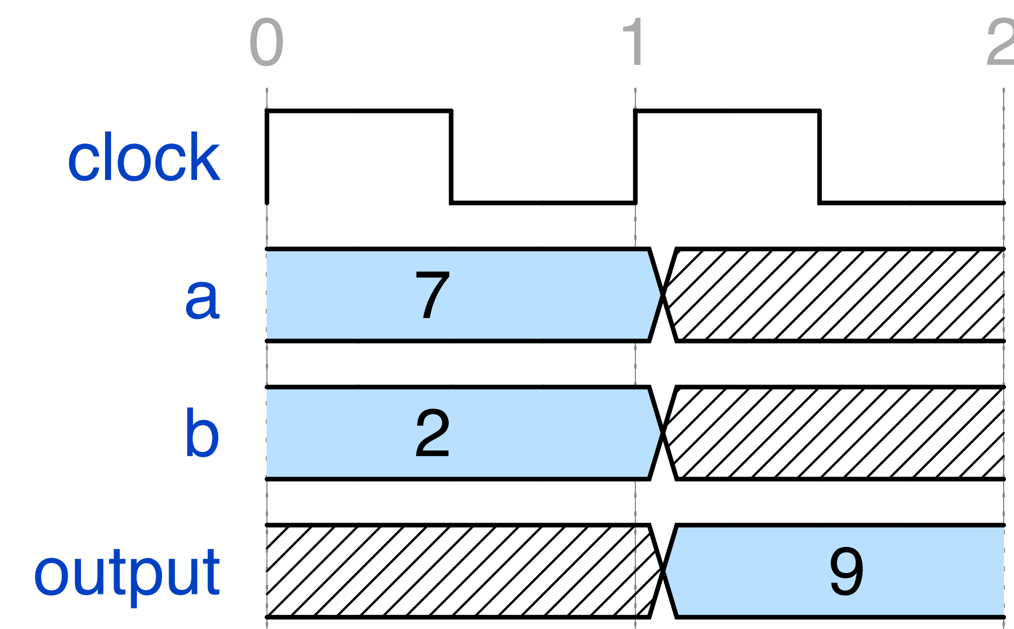
Hardware runs at the level of individual clock cycles & signals,
but we care about high-level **transactions**!

Software

```
fn add(a: u8, b: u8) → u8;
```

Sequential Adder





The abstraction gap in hardware

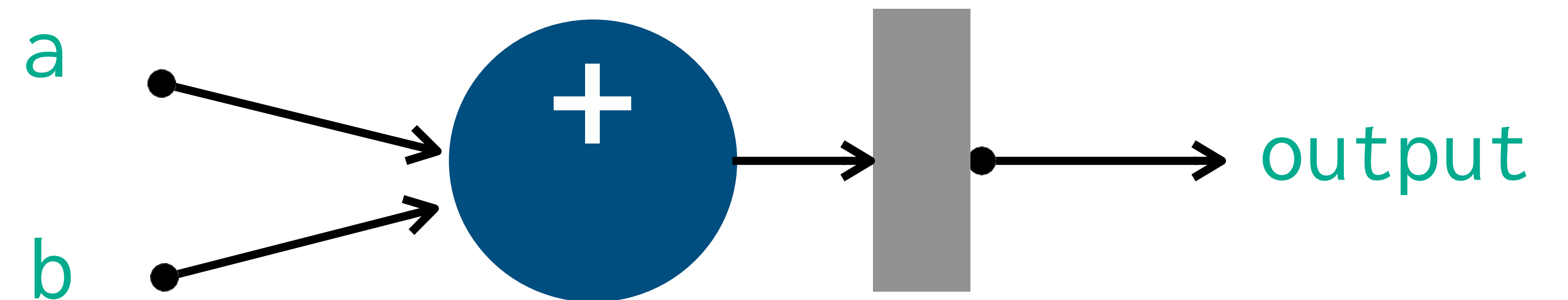
Hardware runs at the level of individual clock cycles & signals,
but we care about high-level **transactions**!

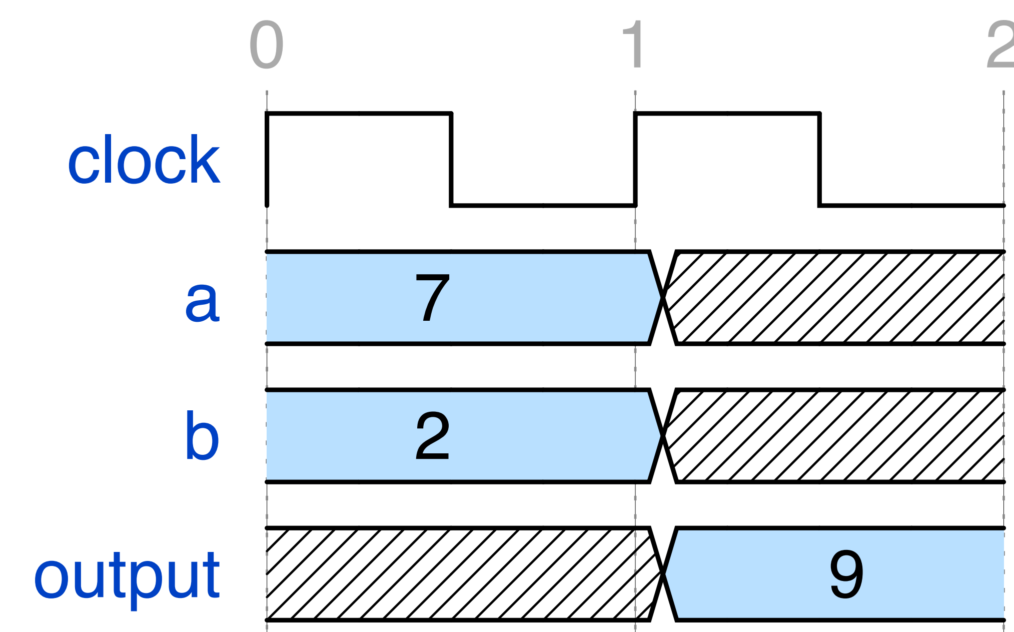
Software

```
fn add(a: u8, b: u8) → u8;
```

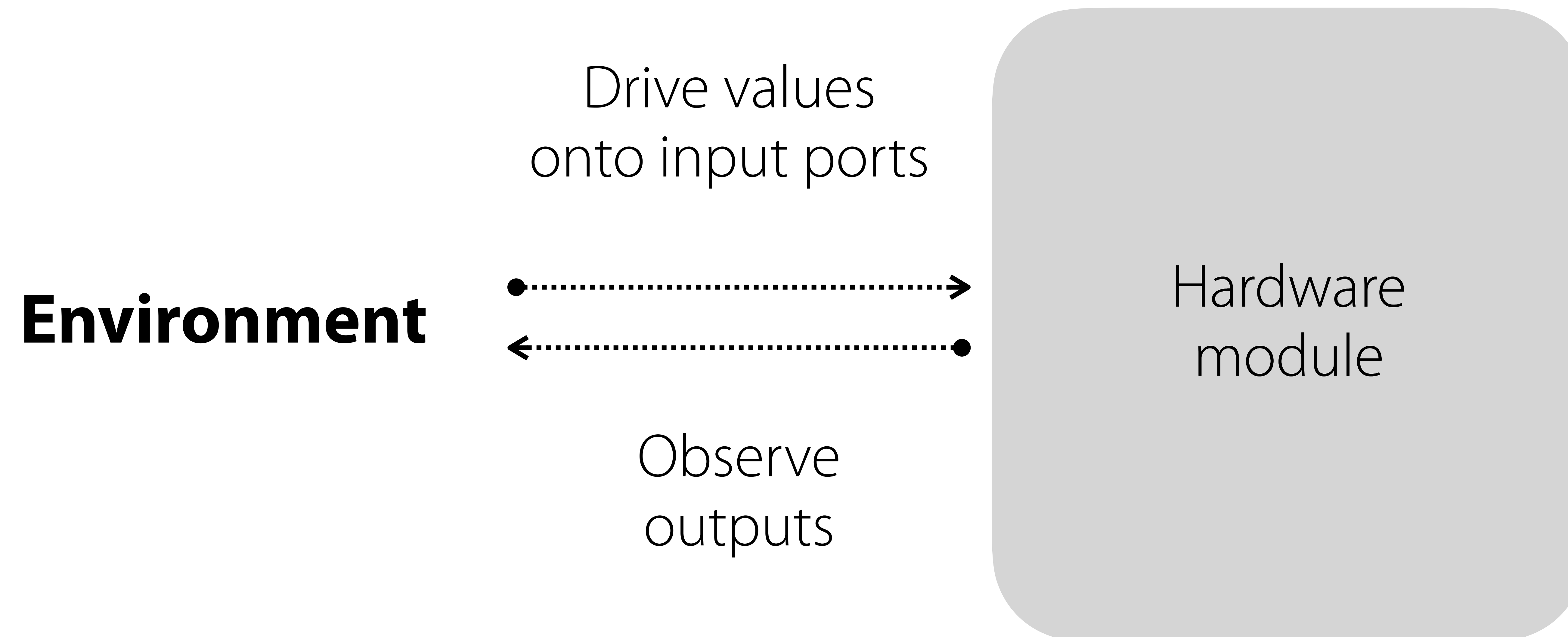
In this talk, we focus on
the I/O communication behavior
of hardware modules,
not their functional correctness

Sequential Adder





Testing hardware is hard



Testing hardware is hard

- **Communication protocol not written down explicitly!**
- Implicitly scattered across ad-hoc tests & assertions

VERILATOR

Environment

Drive values onto input ports

Observe outputs

Hardware module

Debugging hardware is hard

Time

Only notion of time is the **circuit's clock**

(as opposed to a logical program step)

Time	
clk	1
mem_axi_awprot	0
resetn	1
trap	0
trace_valid	1
trace_data	a0000019c
tests_passed	0
mem_axi_wvalid	0
mem_axi_wstrb	0
mem_axi_wready	0
mem_axi_wdata	xxxxxxxx
mem_axi_rvalid	0
mem_axi_rready	0
mem_axi_rdata	0500a023
mem_axi_bvalid	0
mem_axi_bready	0
mem_axi_awvalid	0
irq	00000000
cycle_counter	166

00 ns 1700 ns 1800 ns 1900 ns 2000 ns 2100 ns 2200 ns 2300 ns 2400

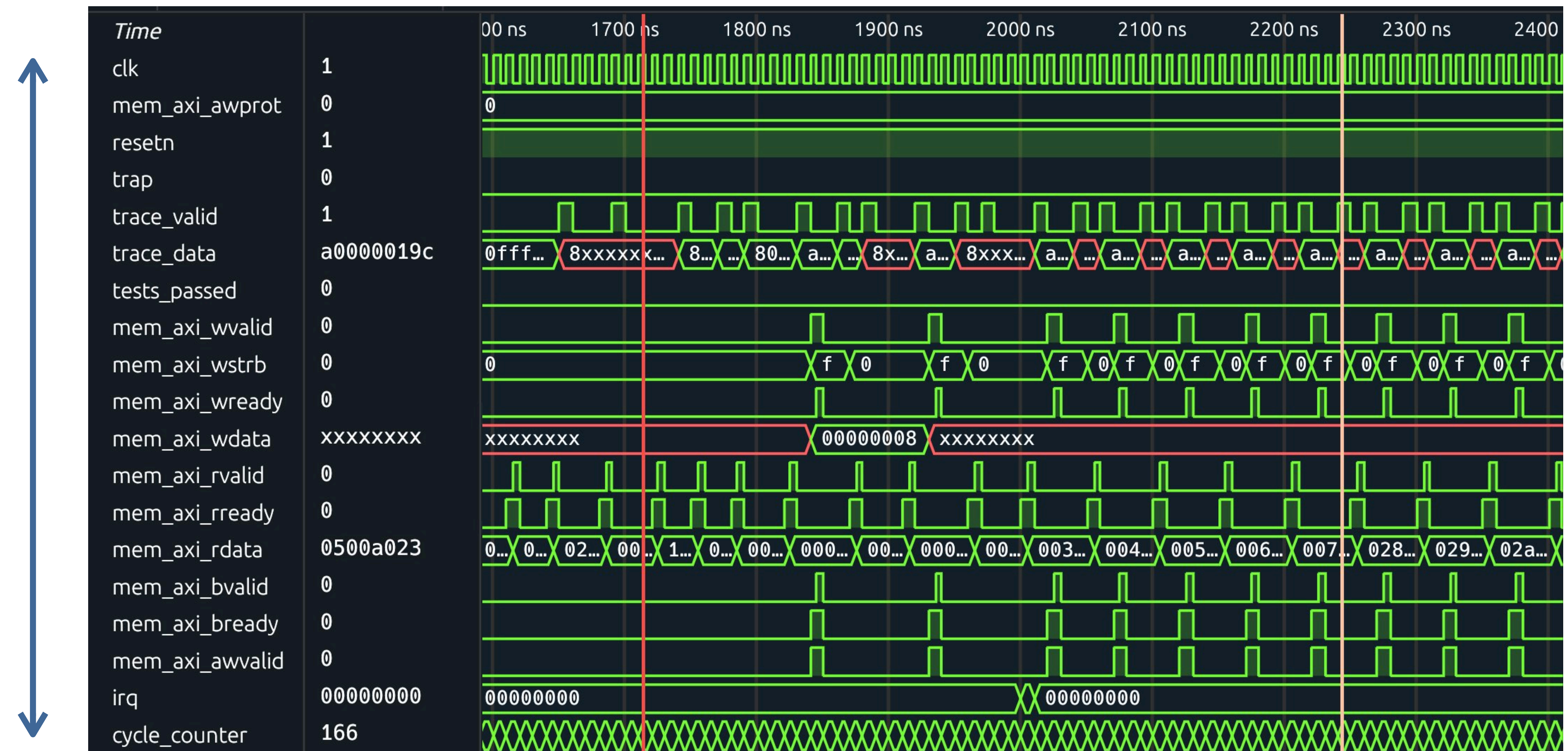
Temporally localizing bugs is hard!

Debugging hardware is hard

All program state flattened into **signals**

(all of which may be equally relevant at any given time)

Spatially localizing bugs is hard!



Every hardware module speaks a protocol

Every hardware module speaks a protocol

Every hardware module speaks a protocol

- Simple handshakes
- On-chip buses (AXI, Wishbone, Avalon)
- Streaming data (AXI-Stream, Avalon-ST)
- Low-speed serial links (UART, SPI, I2C)
- High-speed links (PCIe, CXL)
- …

ARM® AMBA®
Interconnect Standards

CXL

Language features

The usual imperative language features*

Language features

The usual imperative language features*

Assignments

```
input_port := value
```

Language features

The usual imperative language features*

Assignments

```
input_port := value
```

Control flow

```
   if (…) { … }
while (…) { … }
```

Language features

The usual imperative language features*

Assignments

```
input_port := value
```

Control flow

```
   if (…) { … }
while (…) { … }
```

Bit-slicing

```
value[msb:lsb]
```

Language features

The usual imperative language features*

Assignments

```
input_port := value
```

Control flow

```
   if (…) { … }
while (…) { … }
```

Bit-slicing

```
value[msb:lsb]
```

Assertions

```
assert_eq(…, …)
```

Language features

The usual imperative language features*

Assignments

```
input_port := value
```

Control flow

```
   if (…) { … }
while (…) { … }
```

Bit-slicing

```
value[msb:lsb]
```

Assertions

```
assert_eq(…, …)
```

*with restrictions to make the reconstructor tractable

Encoding the Ready-Valid Handshake Rules in Our Protocol

Encoding the Ready-Valid Handshake Rules in Our Protocol

1. Independence

`ready` & `valid` are independent
(Sender can't wait for `ready` to become high before asserting `valid`)

Encoding the Ready-Valid Handshake Rules in Our Protocol

1. Independence

`ready` & `valid` are independent
(Sender can't wait for `ready` to become high before asserting `valid`)

2. Stability

Once `valid` becomes high ,
`valid` & `data` must remain stable (unchanged)
until the handshake completes

Encoding the Ready-Valid Handshake Rules in Our Protocol

1. Independence

`ready` & `valid` are independent
(Sender can't wait for `ready` to become high before asserting `valid`)

2. Stability

Once `valid` becomes high,
`valid` & `data` must remain stable (unchanged) until the handshake completes

1. `valid` asserted independent of `ready`

2. Assignments persist unless overwritten (like SW languages),
i.e. `valid` & `data` remain stable

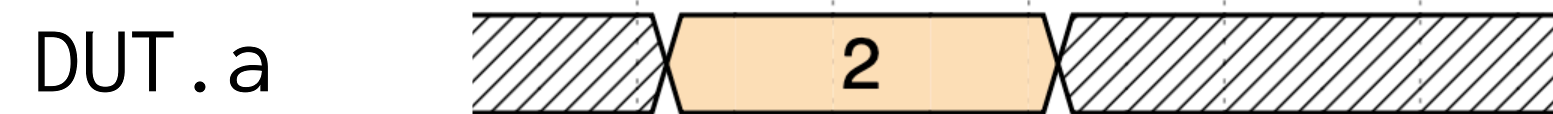
```
prot send_data<DUT: ReadyValid>(data: u8) {
   DUT.data := data;
   DUT.valid := 1;
   while (DUT.ready != 1) {
     step();
   }
   step(); // One cycle for data transfer
}
```

Our protocols serve as executable specs!

Handling Assignments (Variables)

Infer the value of the variable* from the waveform value of the port on the LHS

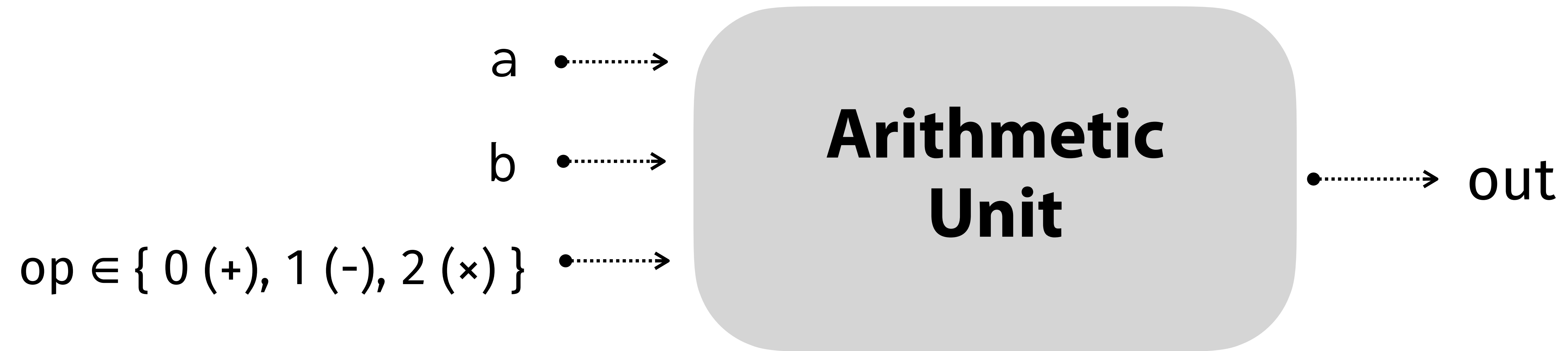
```
prot foo<DUT: …>(var: u8) {
   DUT.a := var;
   …
}
```

DUT.a 

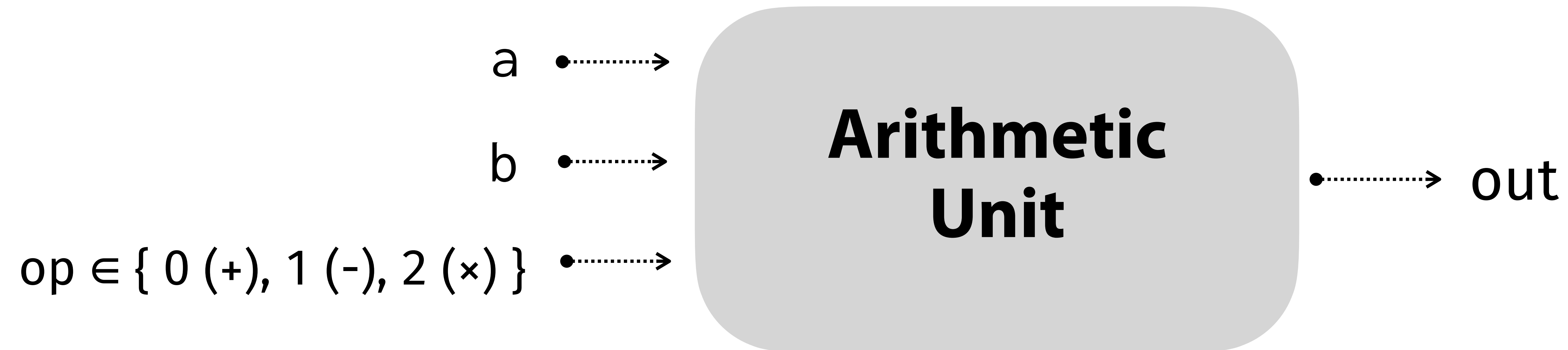
Value of `var` inferred to be 2
(the waveform value for `DUT.a` during the current clock cycle)

*RHS of assignments must be input parameters to functions, which are immutable in our DSL

BFS over possible transactions

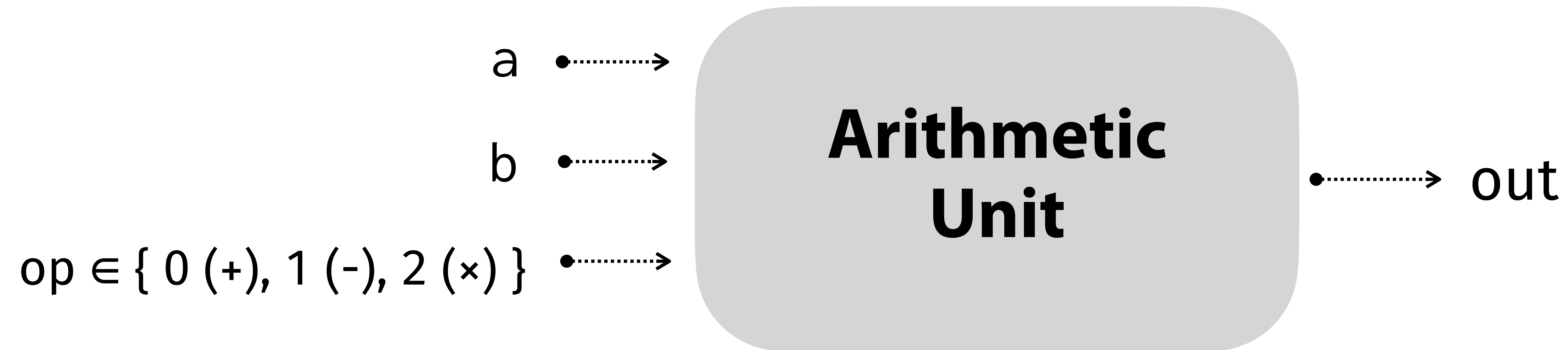


BFS over possible transactions




```
prot add<DUT: …>(
  a: u8, b: u8,
  out: u8
) {
  DUT.op := 0;
  …
}
```

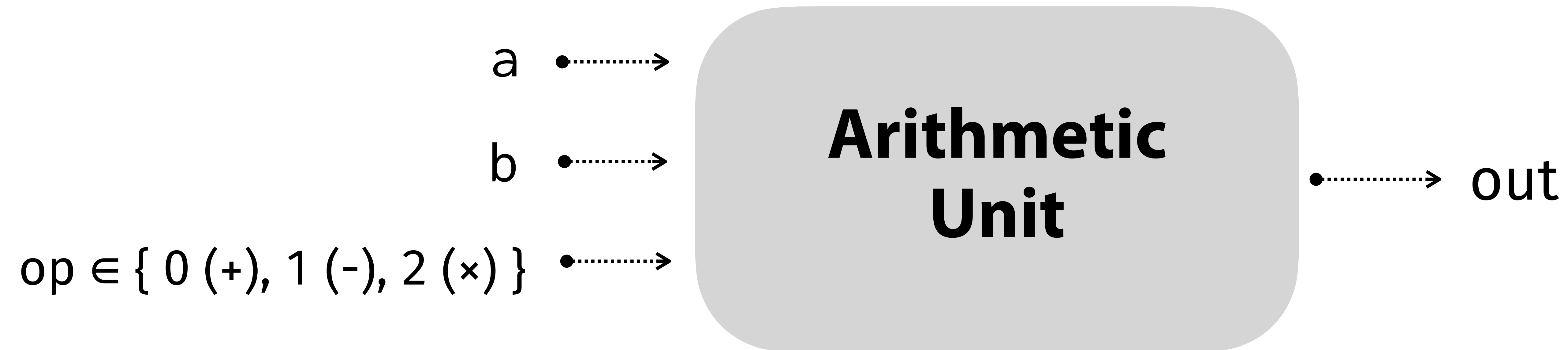
BFS over possible transactions




```
prot add<DUT: …>(
  a: u8, b: u8,
  out: u8
) {
  DUT.op := 0;
  …
}
```

```
prot sub<DUT: …>(
  a: u8, b: u8,
  out: u8
) {
  DUT.op := 1;
  …
}
```

BFS over possible transactions

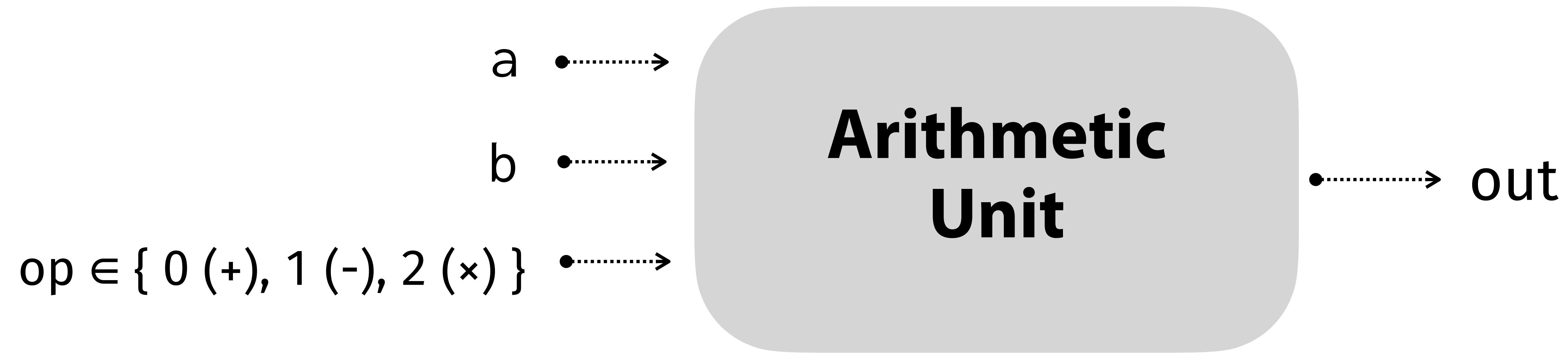



```
prot add<DUT: …>(
  a: u8, b: u8,
  out: u8
) {
  DUT.op := 0;
  …
}
```

```
prot sub<DUT: …>(
  a: u8, b: u8,
  out: u8
) {
  DUT.op := 1;
  …
}
```

```
prot mult<DUT: …>(
  a: u8, b: u8,
  out: u8
) {
  DUT.op := 2;
  …
}
```

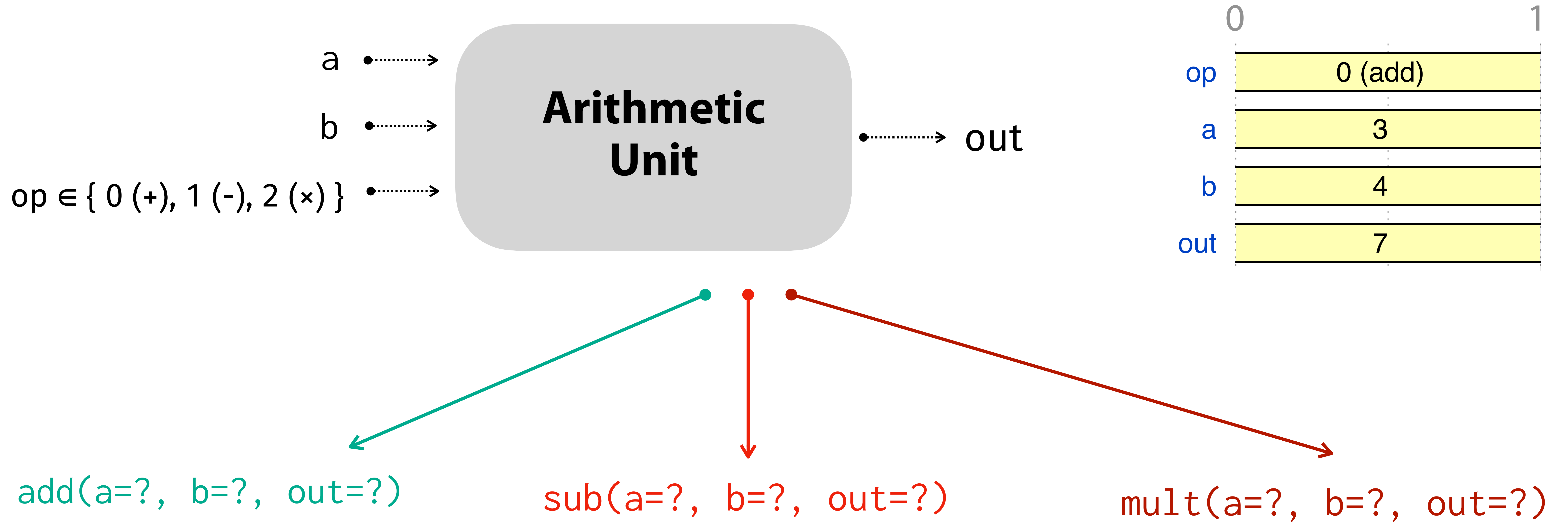
BFS over possible transactions



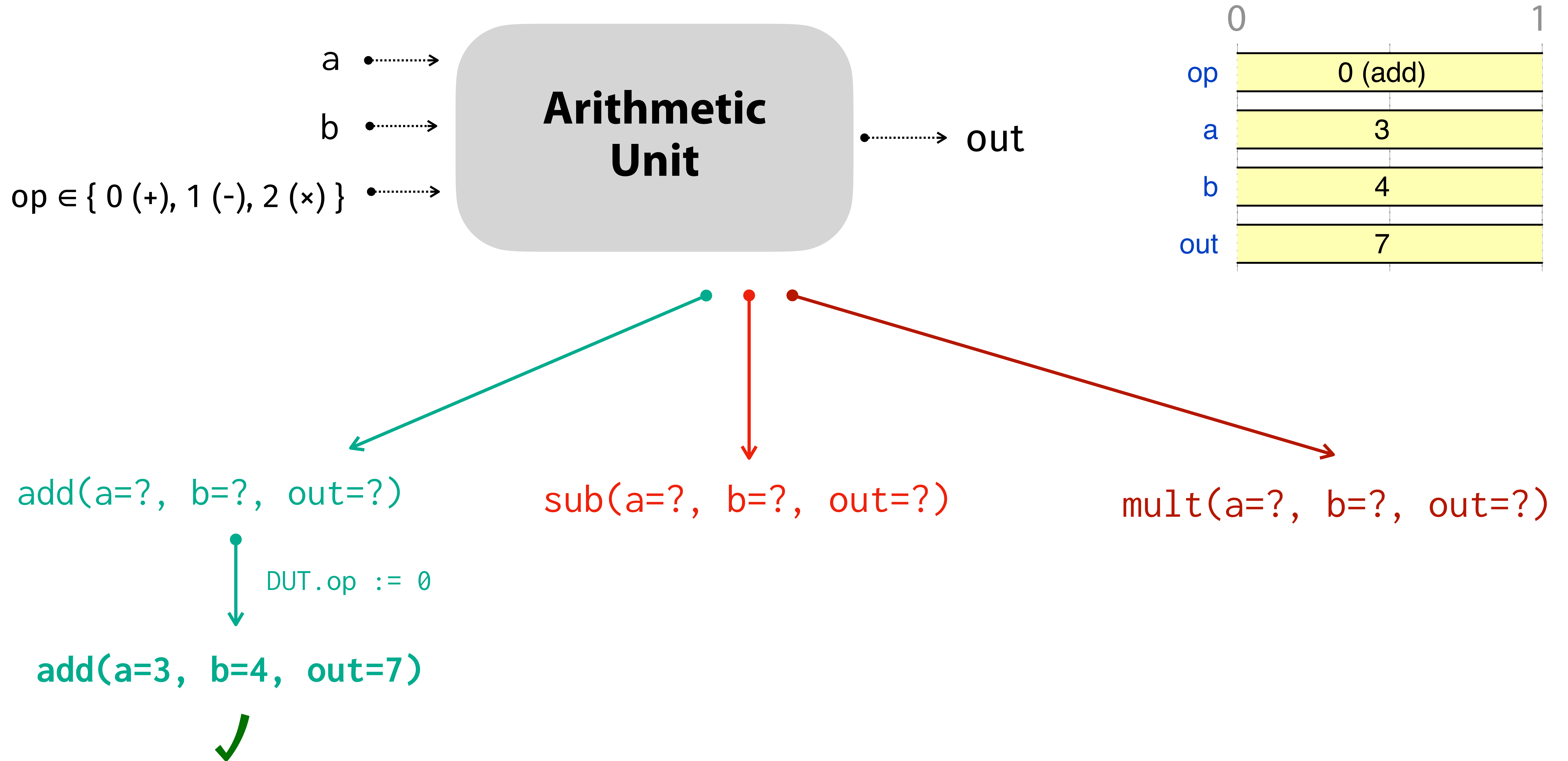
0 1

op 0 (add)
a 3
b 4
out 7

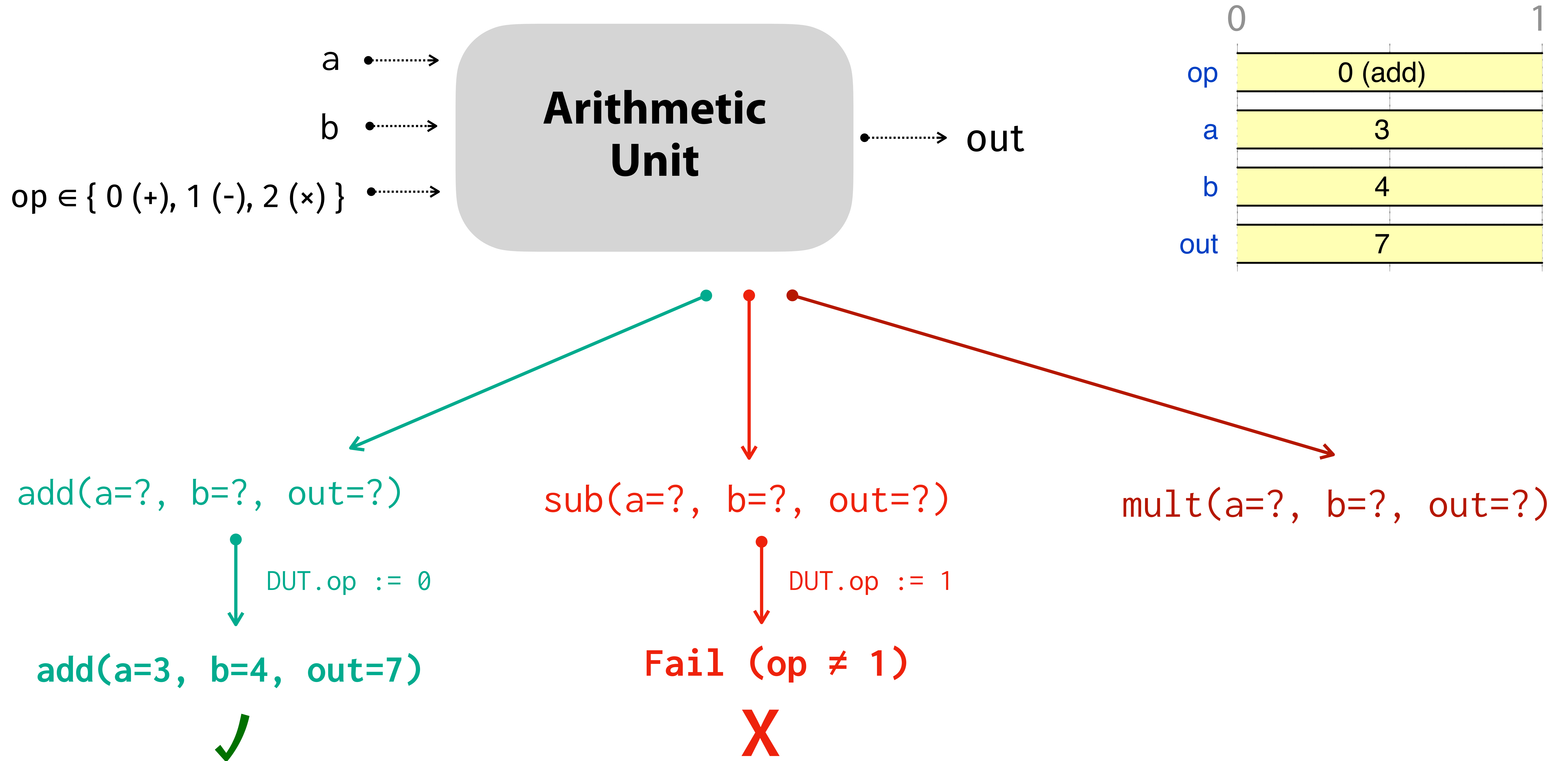
BFS over possible transactions



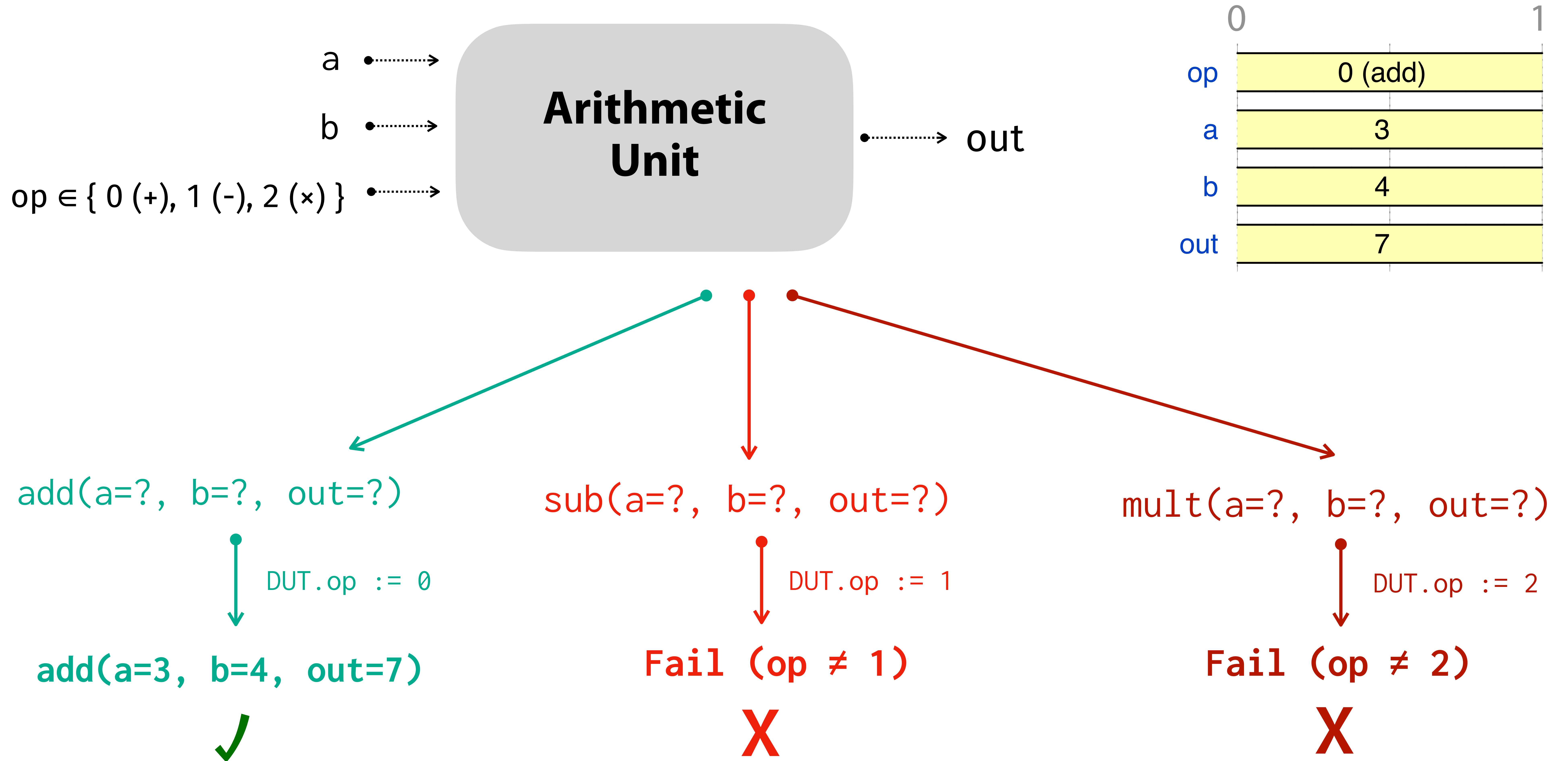
BFS over possible transactions



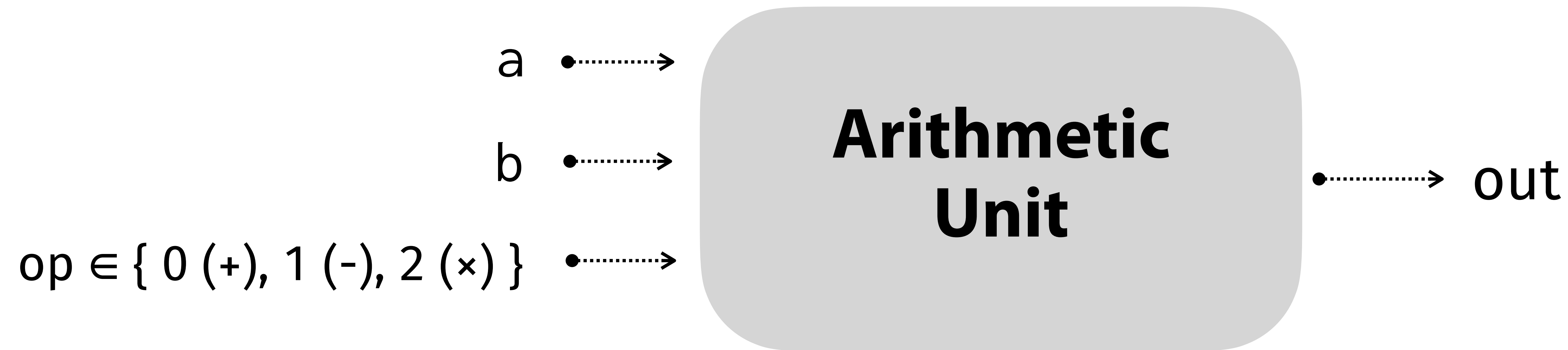
BFS over possible transactions

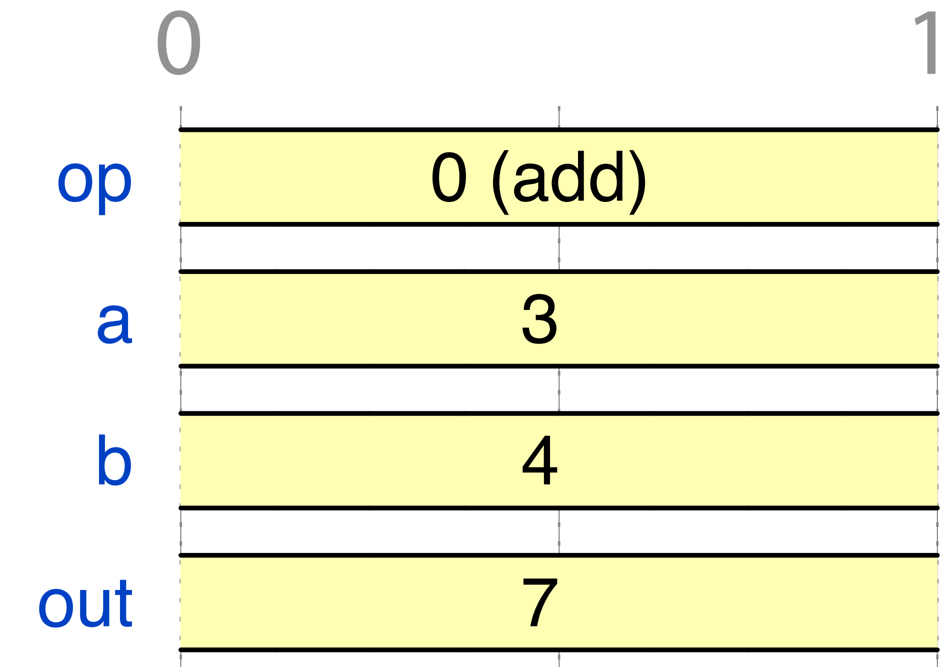


BFS over possible transactions



BFS over possible transactions





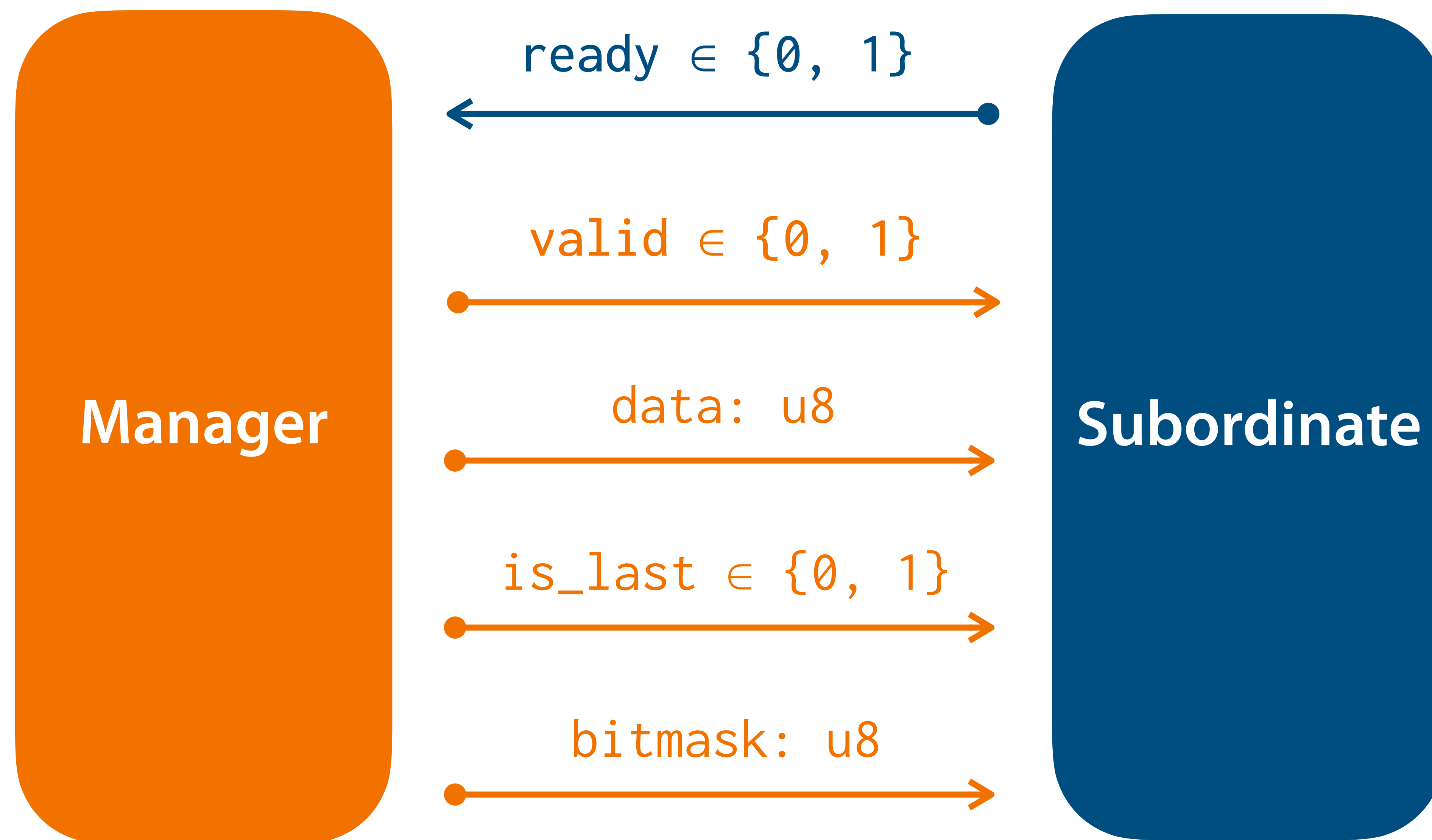

```
op == 0

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

add(a=3, b=4, out=7)
```

Execution paths inconsistent with the waveform are immediately pruned!

Two real-world protocols

AXI-Stream

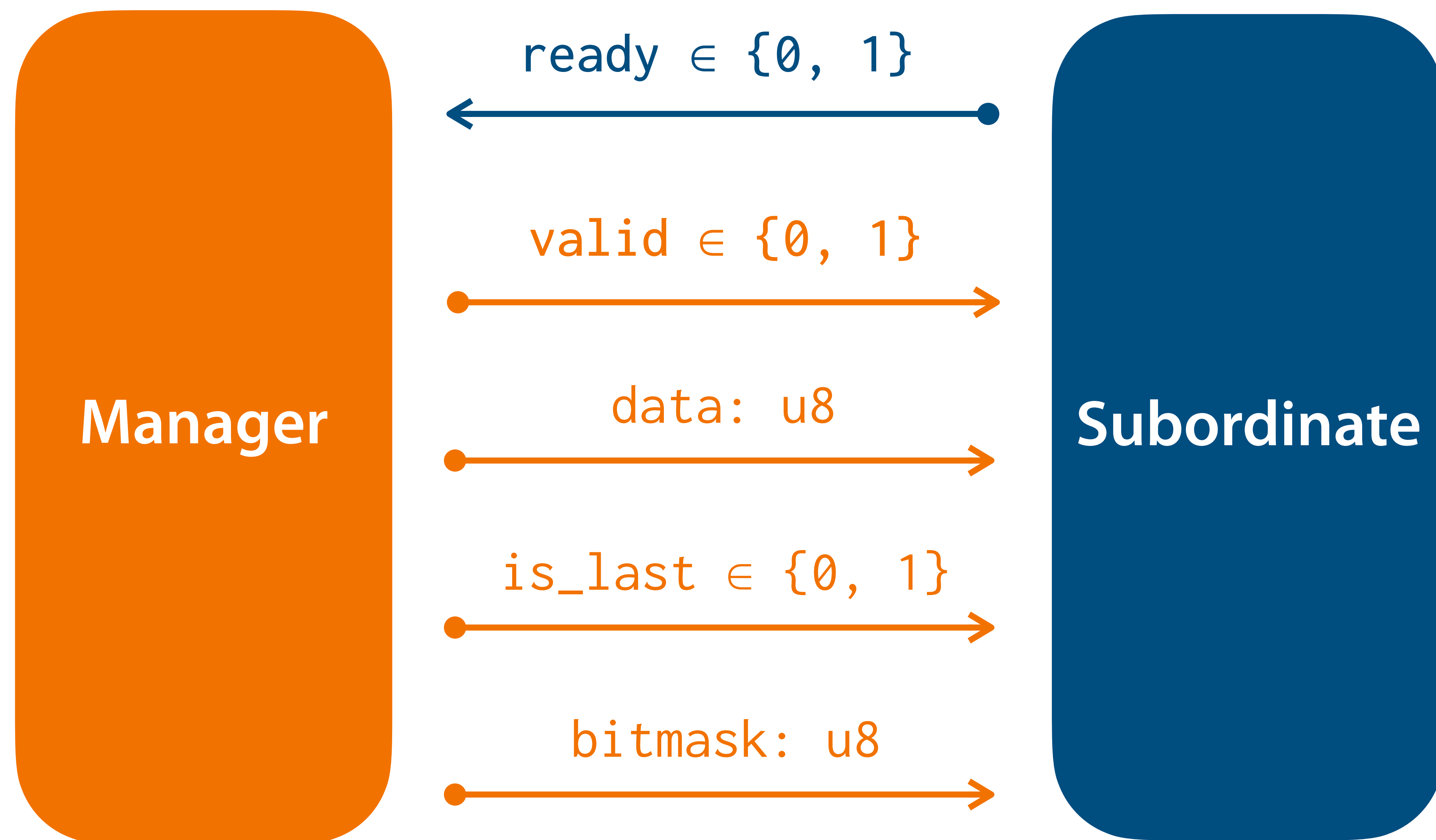
ARM ready-valid interface for stream processing
(e.g. for packets)



Two real-world protocols

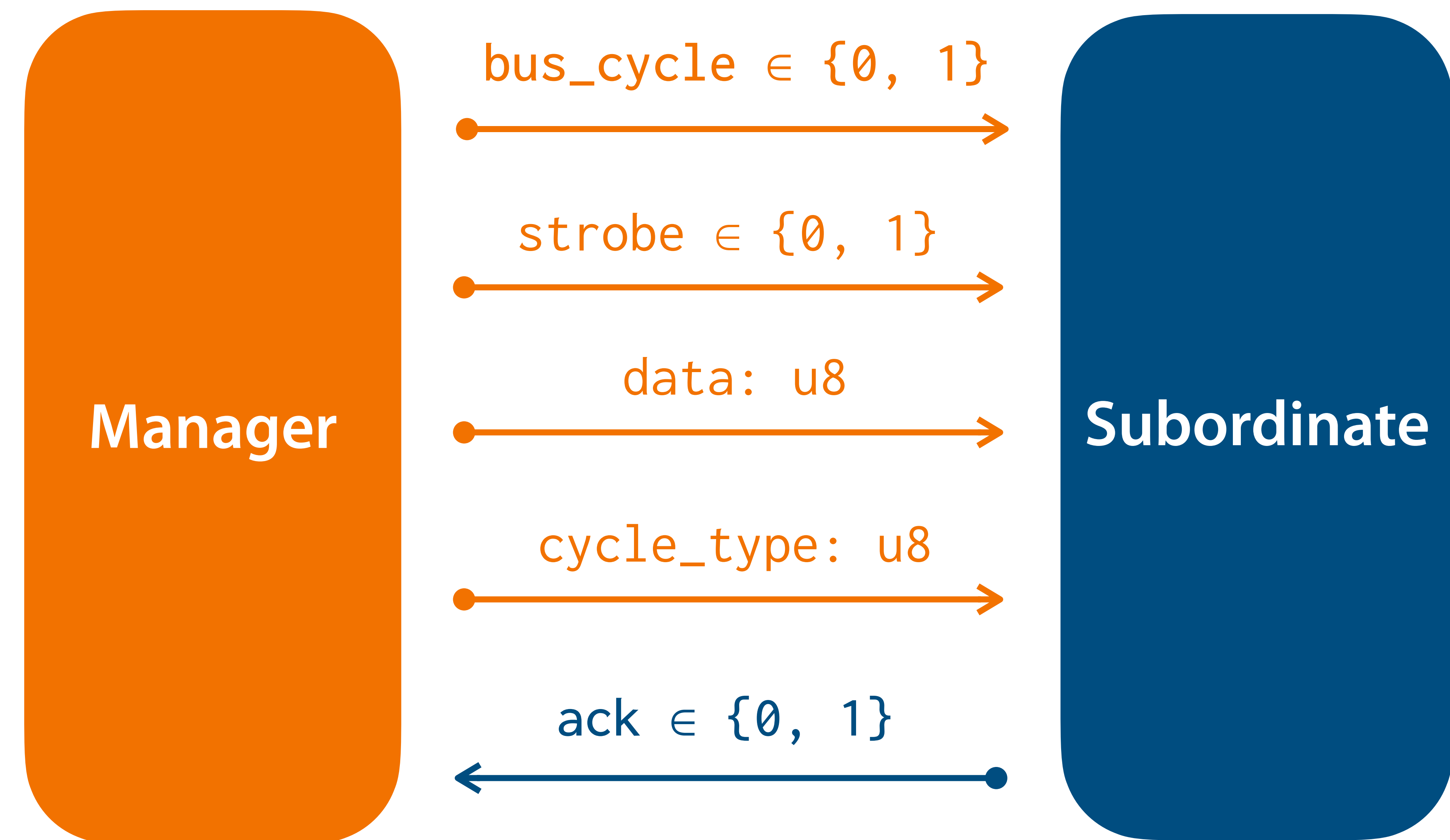
AXI-Stream

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



Wishbone

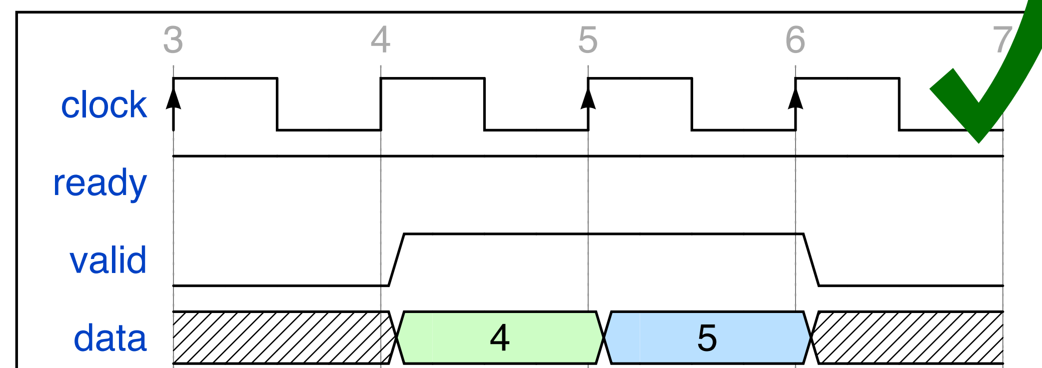
Open-source interconnect for on-chip communication



Protocol bugs in the wild

- Each bug in their benchmark corresponds to a buggy + fixed Verilog module
- We reproduced these bugs, obtained buggy / fixed waveforms, hand-wrote protocol specs in our DSL, then checked if the reconstructor could reveal the bug

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

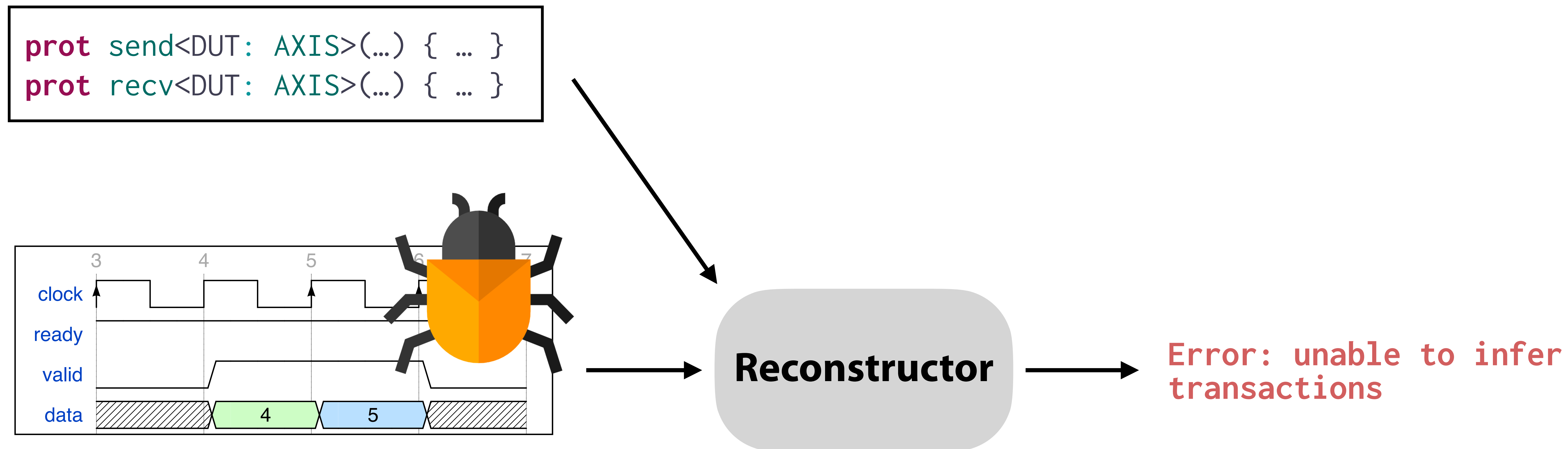


Reconstructor

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

Protocol bugs in the wild

- Each bug in their benchmark corresponds to a buggy + fixed Verilog module
- We reproduced these bugs, obtained buggy / fixed waveforms, hand-wrote protocol specs in our DSL, then checked if the reconstructor could reveal the bug



Avoiding path explosion

Traditional DSE engines suffer from the **path explosion** problem:
Combinatorial explosion in the no. of control-flow paths (e.g. loops)

Avoiding path explosion

Traditional DSE engines suffer from the **path explosion** problem:
Combinatorial explosion in the no. of control-flow paths (e.g. loops)

To avoid this, our DSL is designed so that non-deterministic behavior arises in a structured manner:

Avoiding path explosion

Traditional DSE engines suffer from the **path explosion** problem:
Combinatorial explosion in the no. of control-flow paths (e.g. loops)

To avoid this, our DSL is designed so that non-deterministic behavior arises in a structured manner:

```
fork()
```

Avoiding path explosion

Traditional DSE engines suffer from the **path explosion** problem:
Combinatorial explosion in the no. of control-flow paths (e.g. loops)

To avoid this, our DSL is designed so that non-deterministic behavior arises in a structured manner:

`fork()`

Spawn one new execution path for each user-supplied protocol & perform BFS (c.f. ALU example earlier)

Avoiding path explosion

Traditional DSE engines suffer from the **path explosion** problem:
Combinatorial explosion in the no. of control-flow paths (e.g. loops)

To avoid this, our DSL is designed so that non-deterministic behavior arises in a structured manner:

`fork()`

Spawn one new execution path for each user-supplied protocol & perform BFS (c.f. ALU example earlier)

`repeat` loops

Avoiding path explosion

Traditional DSE engines suffer from the **path explosion** problem:
Combinatorial explosion in the no. of control-flow paths (e.g. loops)

To avoid this, our DSL is designed so that non-deterministic behavior arises in a structured manner:

`fork()`

Spawn one new execution path for each user-supplied protocol & perform BFS (c.f. ALU example earlier)

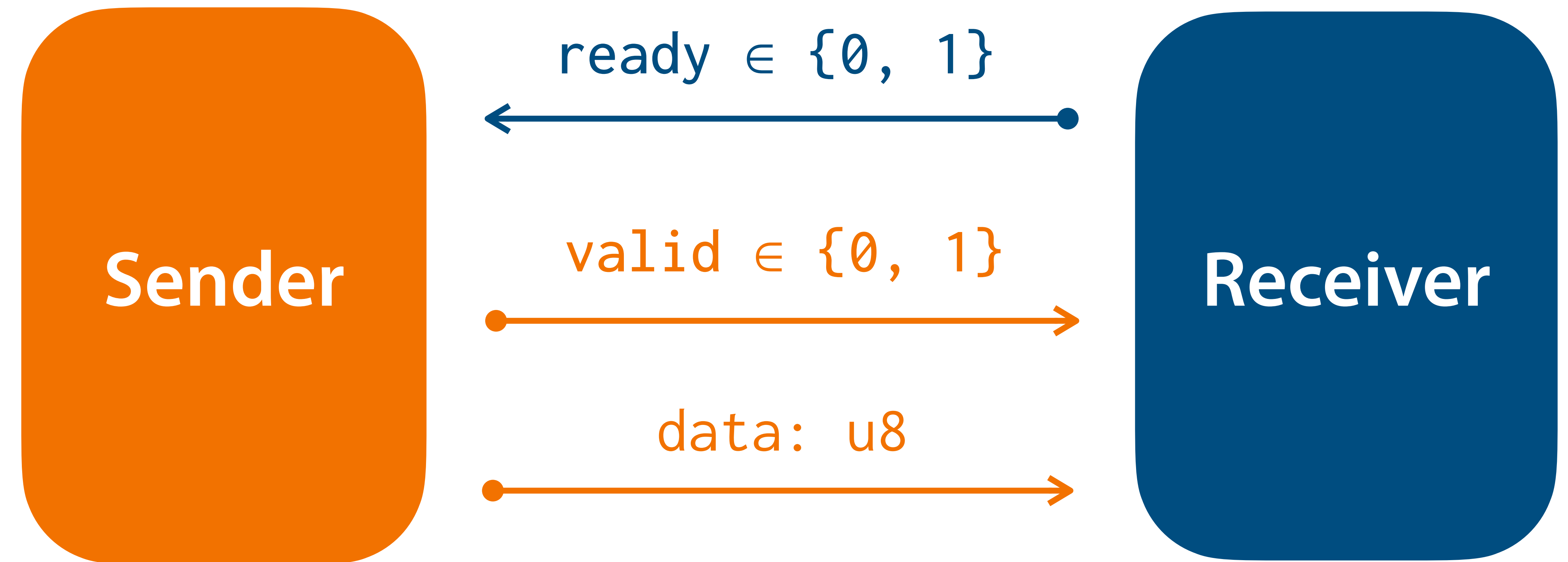
`repeat` loops

(Next slide)

Ready-Valid Handshake, redux

(now from the receiver's view)

```
prot recv<DUT: ReadyValid>(
  data: u8, n: uint
) {
  // Wait for valid data
  while (DUT.valid != 1) { step() }
  assert_eq(DUT.data, data);
  DUT.ready := 0;

  // Apply backpressure for n cycles
  repeat n iterations {
    step();
    assert_eq(DUT.valid, 1);
    assert_eq(DUT.data, data);
  }

  DUT.ready := 1;
  step();
}
```



A **`repeat`** loop executes its body for a fixed no. of iterations

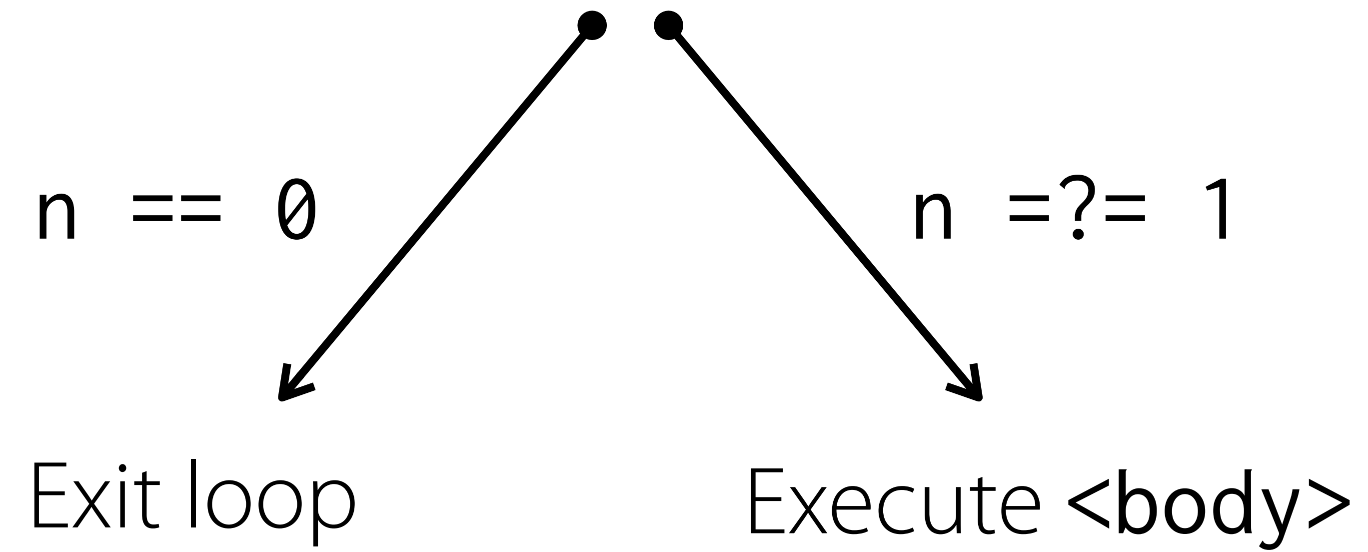
Receiver uses a **`repeat`** loop to exert backpressure (signal unreadiness) for **n** cycles

$\Longrightarrow$ reconstructor must infer **n** based on waveform

Handling `repeat` loops in the reconstructor

```
repeat n iterations {
  <body>
}
…
```

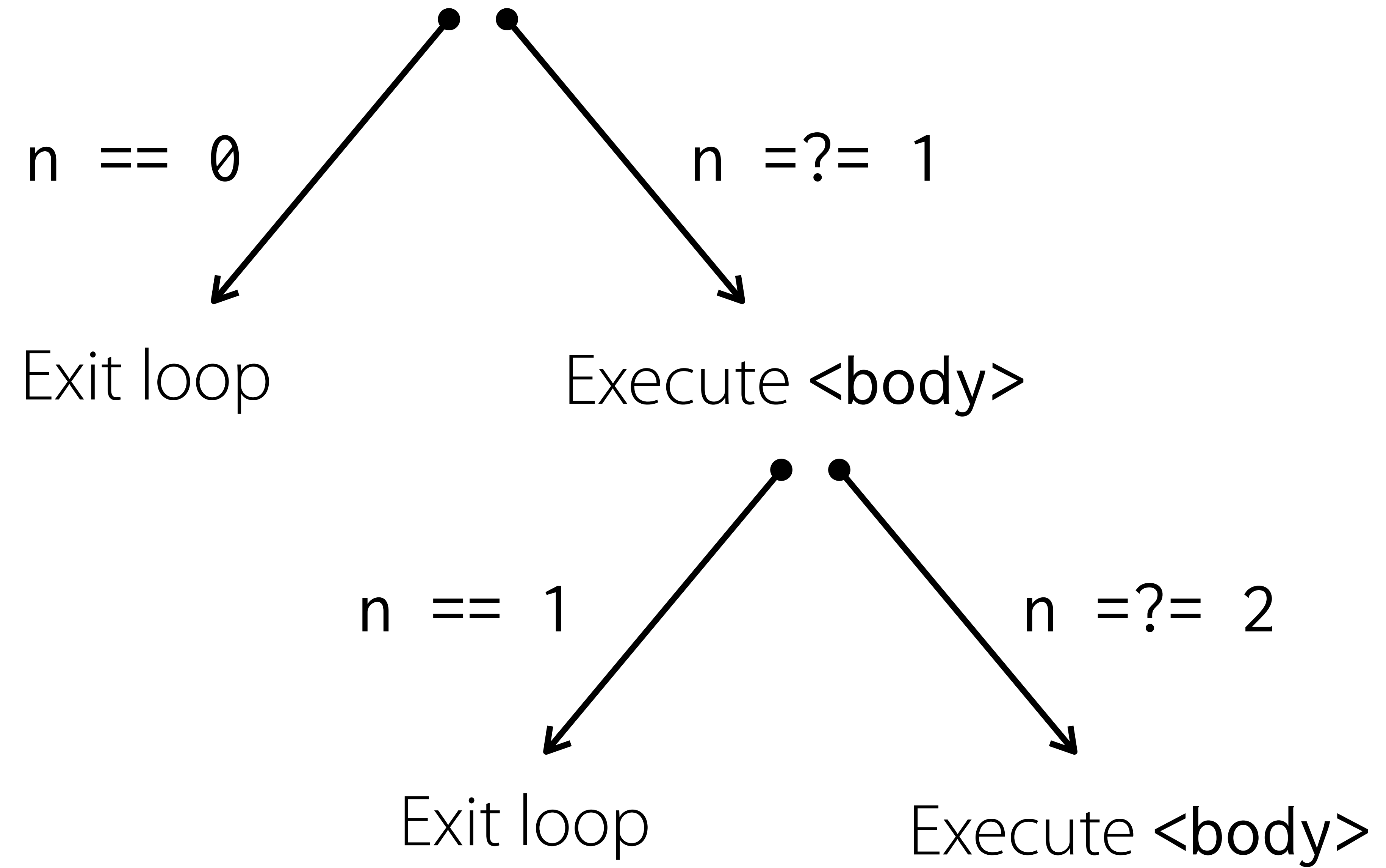
Handling repeat loops in the reconstructor




```
repeat n iterations {
  <body>
}
...
```

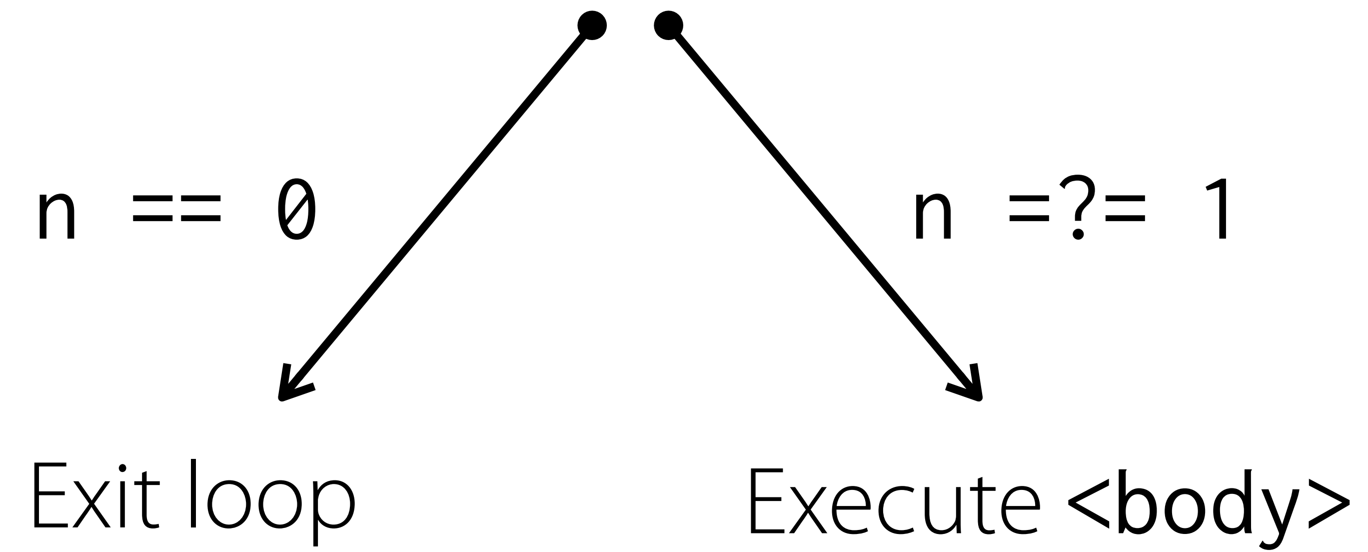
Handling repeat loops in the reconstructor

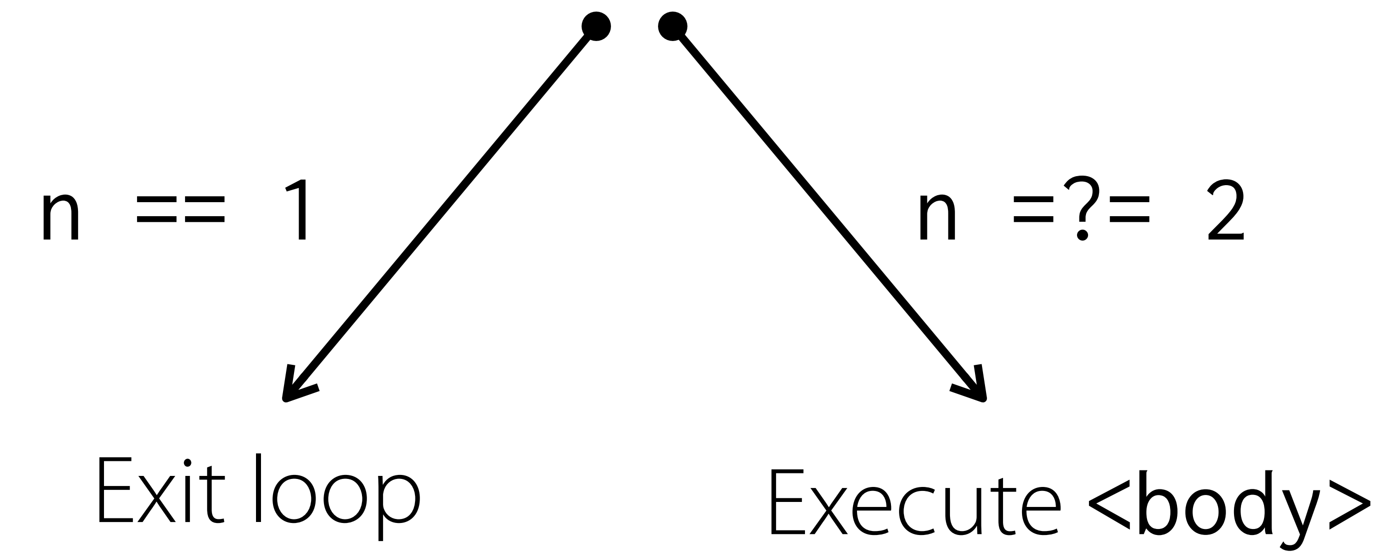
```
repeat n iterations {
  <body>
}
...
```

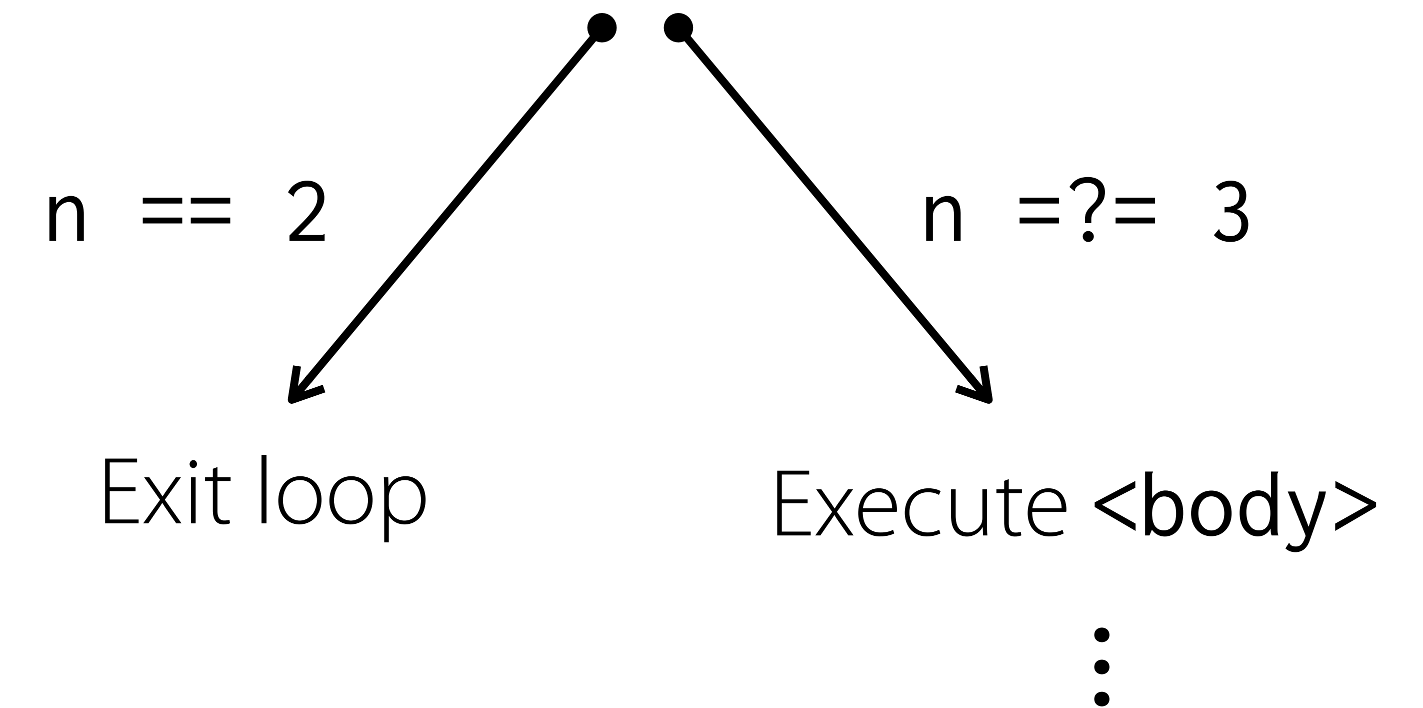


Handling repeat loops in the reconstructor

```
repeat n iterations {
  <body>
}
...
```

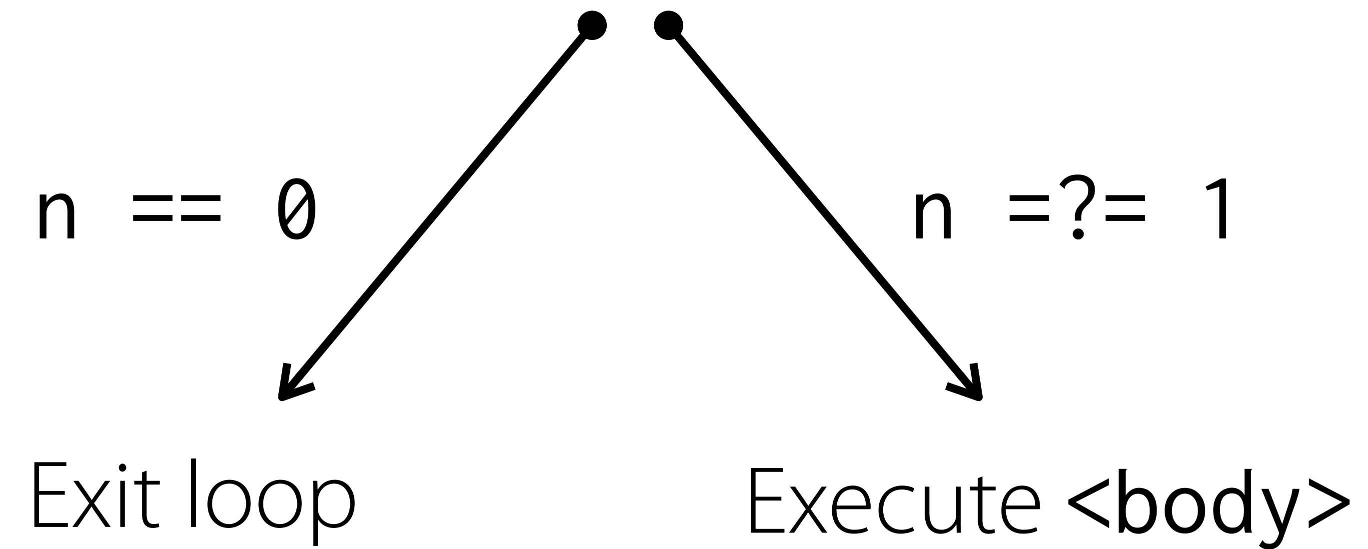


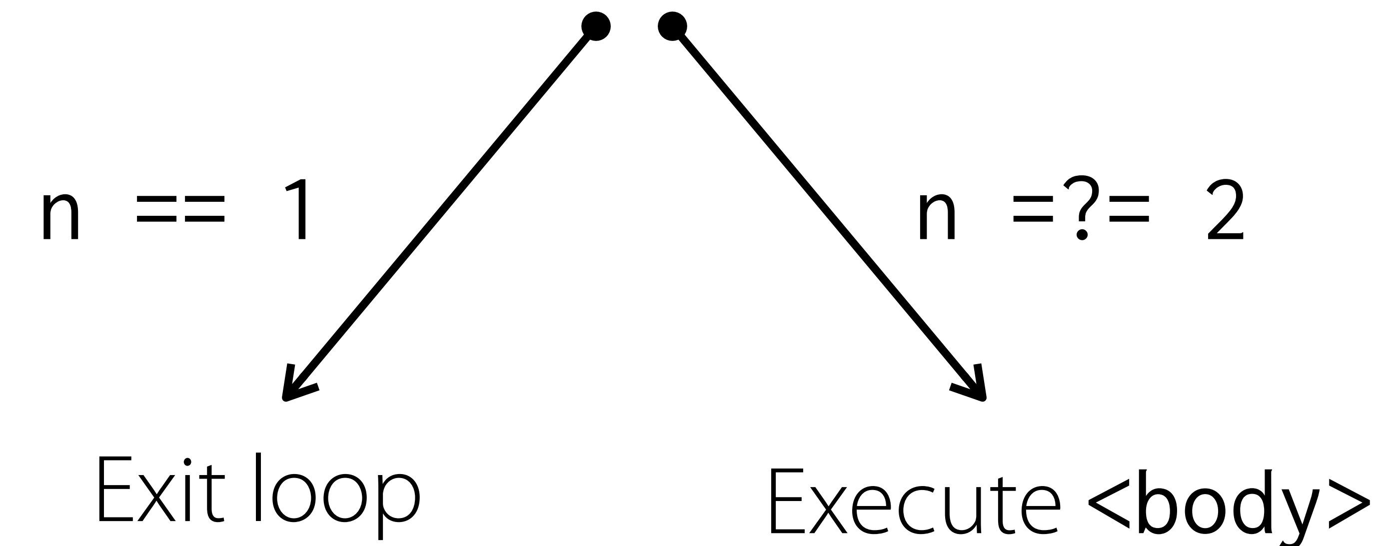


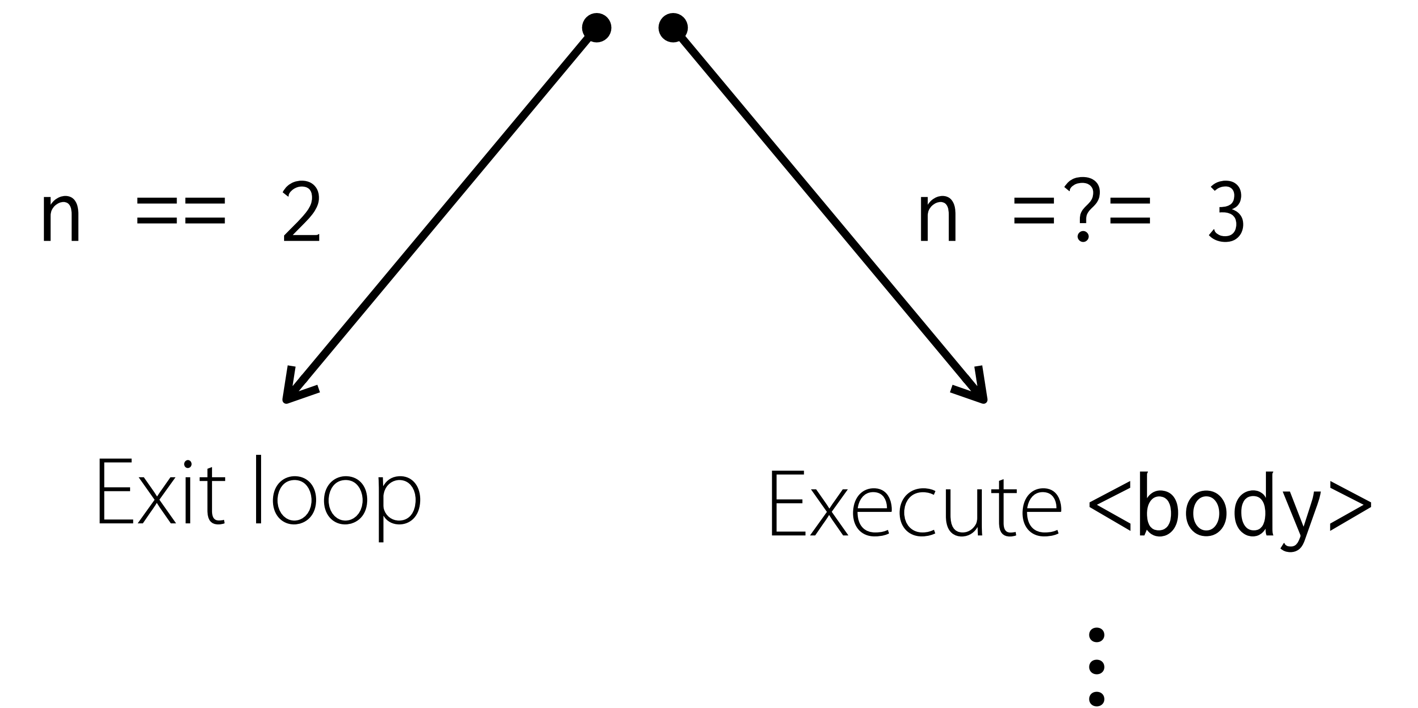


Handling repeat loops in the reconstructor

```
repeat n iterations {
  <body>
}
...
```







Most of these execution paths are pruned once assignments / assertions inconsistent with the waveform are detected

Aside: Dynamic Symbolic Execution (DSE)

Aside: Dynamic Symbolic Execution (DSE)

- Run programs with symbolic inputs instead of concrete ones

Aside: Dynamic Symbolic Execution (DSE)

- Run programs with symbolic inputs instead of concrete ones
- Explores all control-flow paths through a program

Aside: Dynamic Symbolic Execution (DSE)

- Run programs with symbolic inputs instead of concrete ones
- Explores all control-flow paths through a program
- For each path: keep track of branching conditions that are true, then use SMT solvers to determine if constraints are satisfiable

Aside: Dynamic Symbolic Execution (DSE)

- Run programs with symbolic inputs instead of concrete ones
- Explores all control-flow paths through a program
- For each path: keep track of branching conditions that are true, then use SMT solvers to determine if constraints are satisfiable
- Generalizes testing!

Aside: Dynamic Symbolic Execution (DSE)

- Run programs with symbolic inputs instead of concrete ones
- Explores all control-flow paths through a program
- For each path: keep track of branching conditions that are true, then use SMT solvers to determine if constraints are satisfiable
- Generalizes testing!

Popularized by KLEE / CUTE
[Godefroid et al. '05, Cadar et al. '08, …]

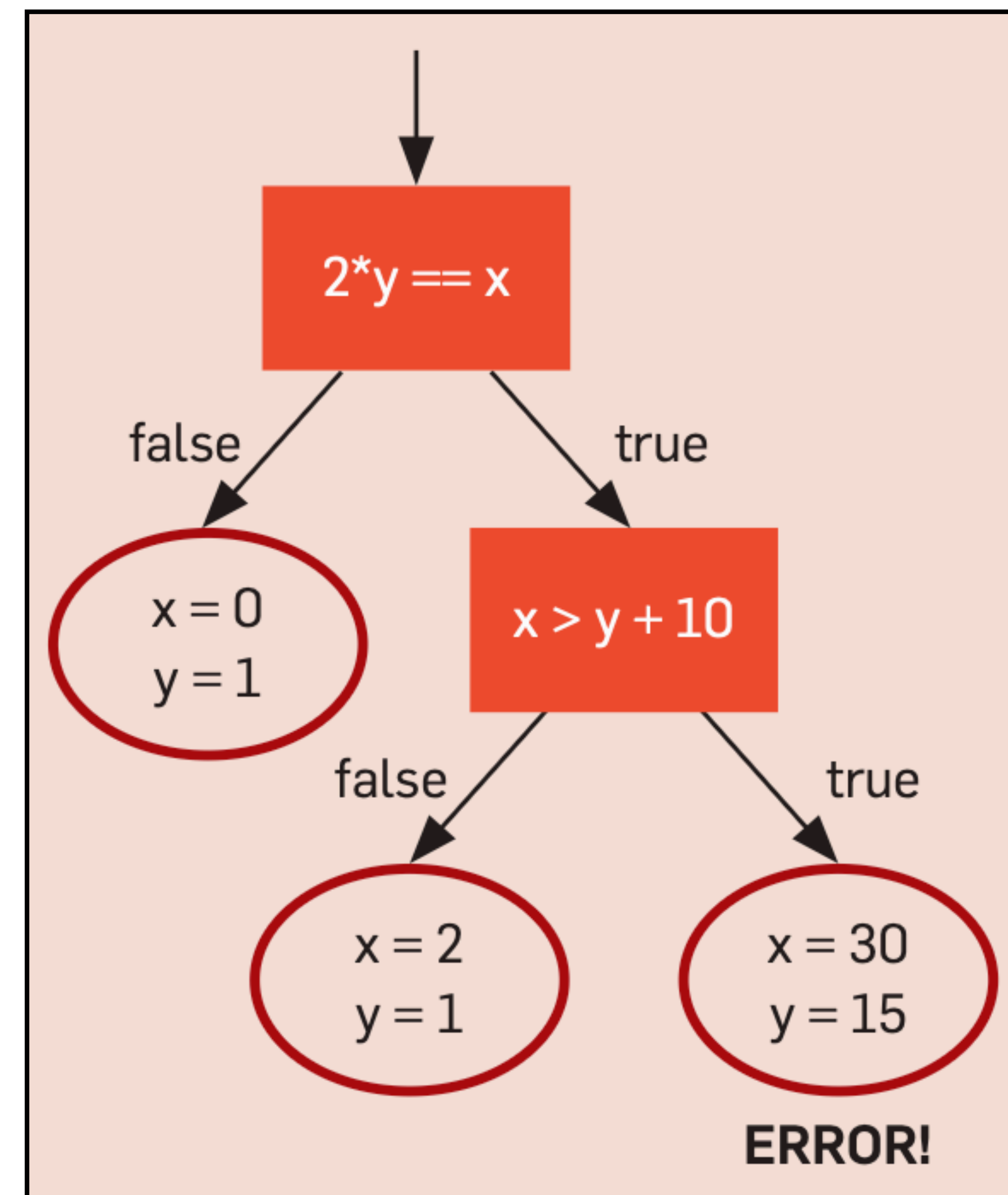
Aside: Dynamic Symbolic Execution (DSE)

- Run programs with symbolic inputs instead of concrete ones
- Explores all control-flow paths through a program
- For each path: keep track of branching conditions that are true, then use SMT solvers to determine if constraints are satisfiable
- Generalizes testing!

Popularized by KLEE / CUTE
[Godefroid et al. '05, Cadar et al. '08, …]

```
void foo(int x, int y) {
  if (2 * y == x) {
    if (x > y + 10) {
      ERROR;
    }
  }
}
```

Example from Cadar & Sen (CACM '13)



Combinational dependency tracking

Goal: interpreter execution should appear deterministic to the user

Invariant: within the same clock cycle, all observations (reads) of an output port `DUT.out` produce the same value

(Otherwise, the value of `DUT.out` during the cycle would be ill-defined!)

Well-formedness rules

Goal: interpreter execution should appear deterministic

Invariant: at most one transaction begins each cycle
(i.e. transactions can be strictly ordered by start time)

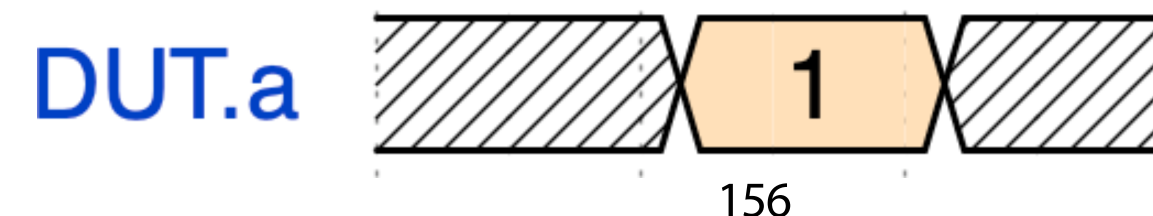
A protocol can `fork()` at most once

`step()` must precede `fork()`

Well-formedness rules

- Protocols must end with `step()`
  - Ensures protocols finish at cycle boundaries
  - Signals remain driven onto DUT ports until the end of the last cycle, after which they're released Avoids needing to reason about what happens "in the remainder of a cycle" if a protocol finishes mid-cycle

Handling bit-slices in the reconstructor

Program statement	Waveform value for `DUT.a`	State of `x` in environment (Changes highlighted in blue)
start of program (`x` uninitialized)	N/A	`value: 0b0000` `known_bits: 0b0000`
`DUT.a := x[0]`		`value: 0b0000` `known_bits: 0b0001`
`DUT.a := x[1]`		`value: 0b0001` `known_bits: 0b0011`