


Capra lab @ Cornell

Specifying Hardware Communication as Programs

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

New Jersey Programming Languages & Systems Seminar
May 22, 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

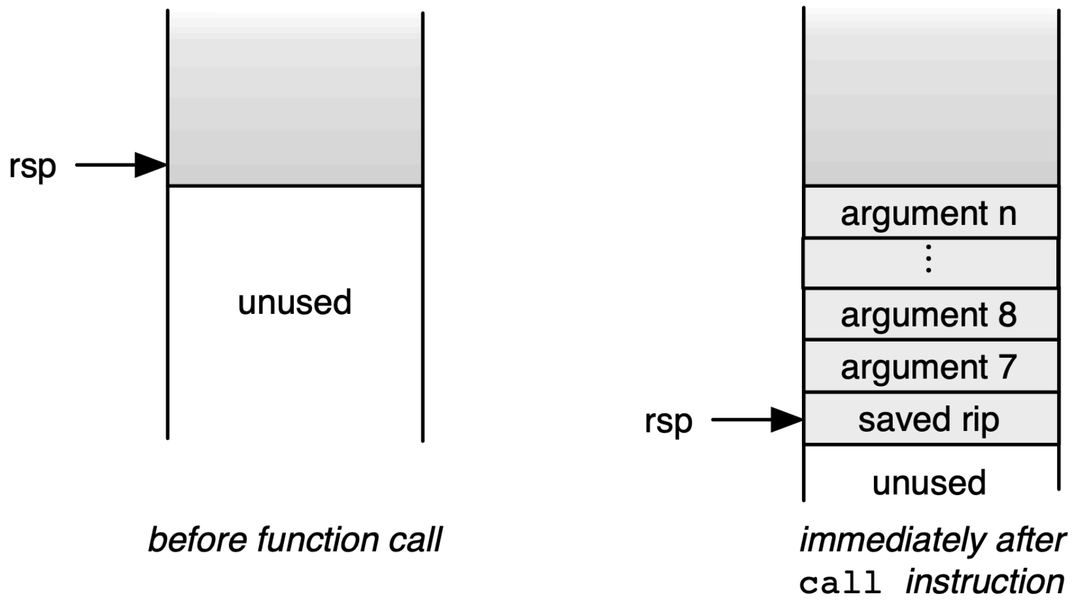


Figure from Andrew Myers

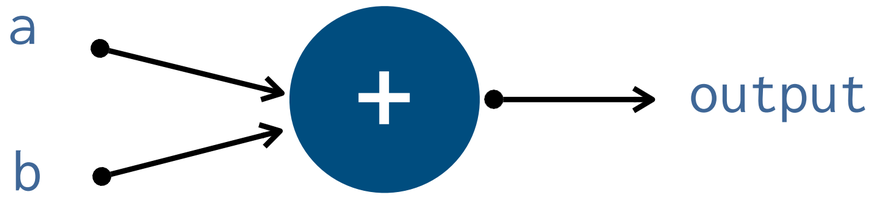
Calling Conventions in *Hardware*?

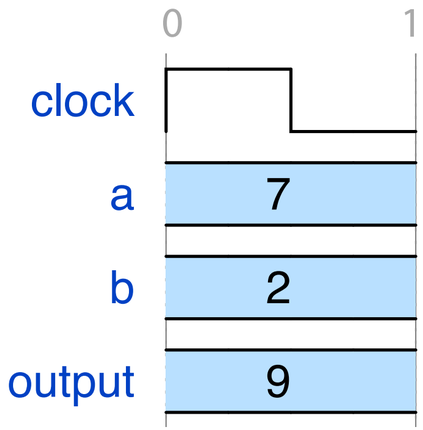
Calling Conventions in *Hardware*?

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

“Calling” an adder in hardware

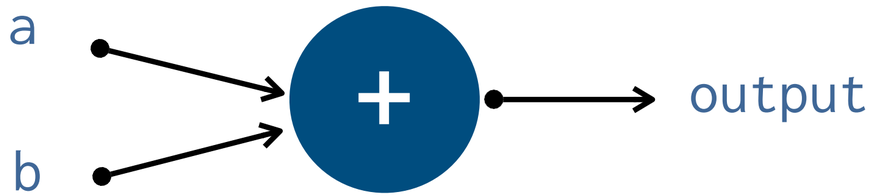
Combinational Adder

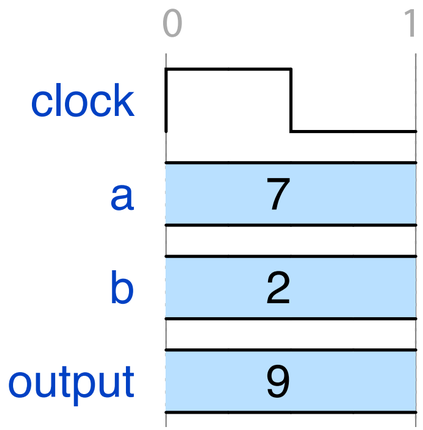




“Calling” an adder in hardware

Combinational Adder



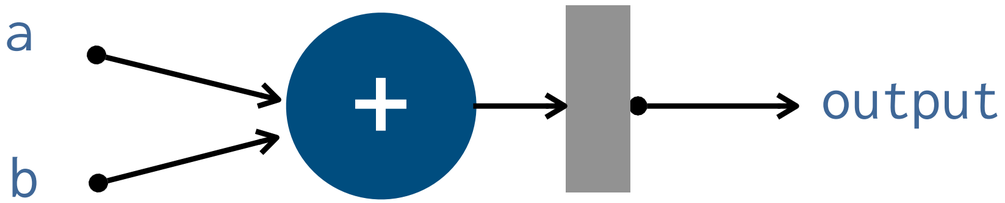


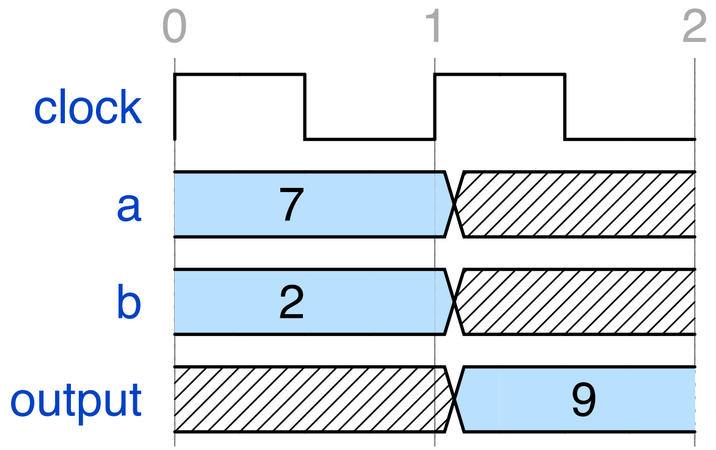
“If I drive `a = 7`, `b = 2`,

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

“Calling” an adder in hardware

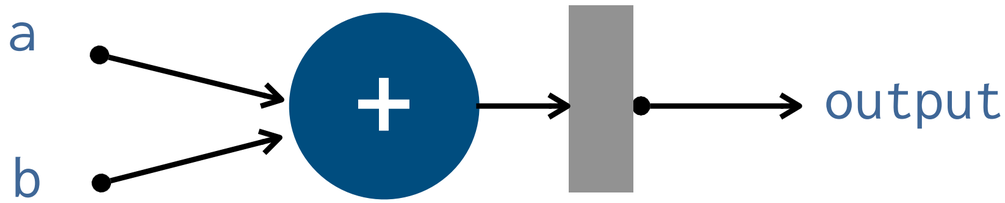
Sequential Adder

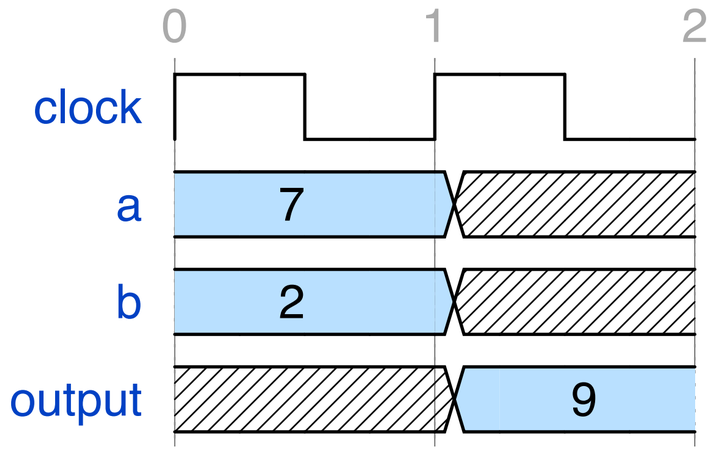




“Calling” an adder in hardware

Sequential Adder





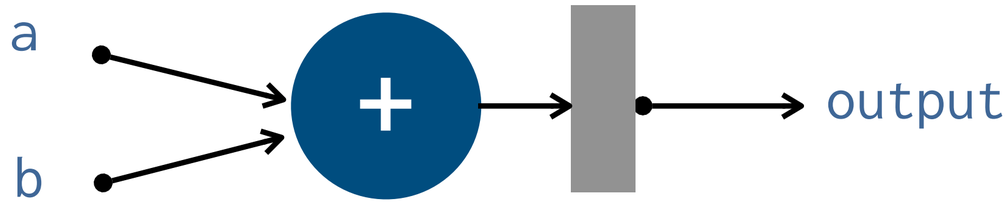
“If I drive `a = 7`, `b = 2`,

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

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

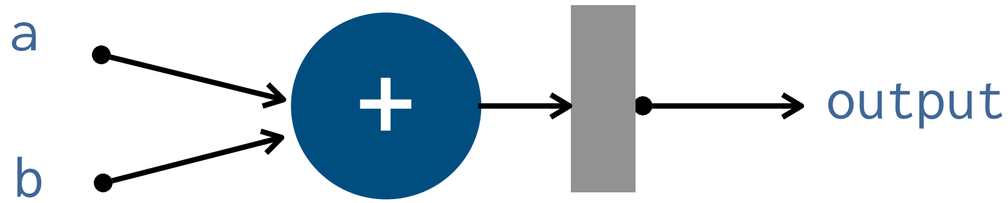
The abstraction gap in hardware

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**!

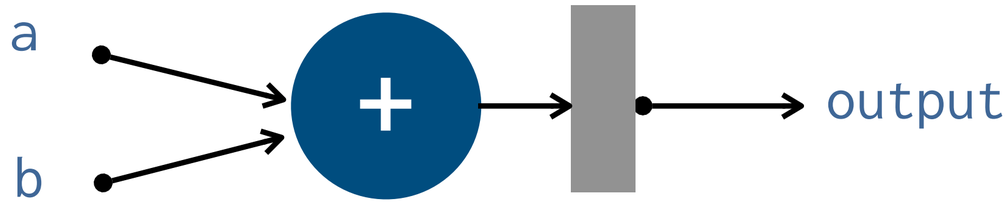


Transactions

"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)

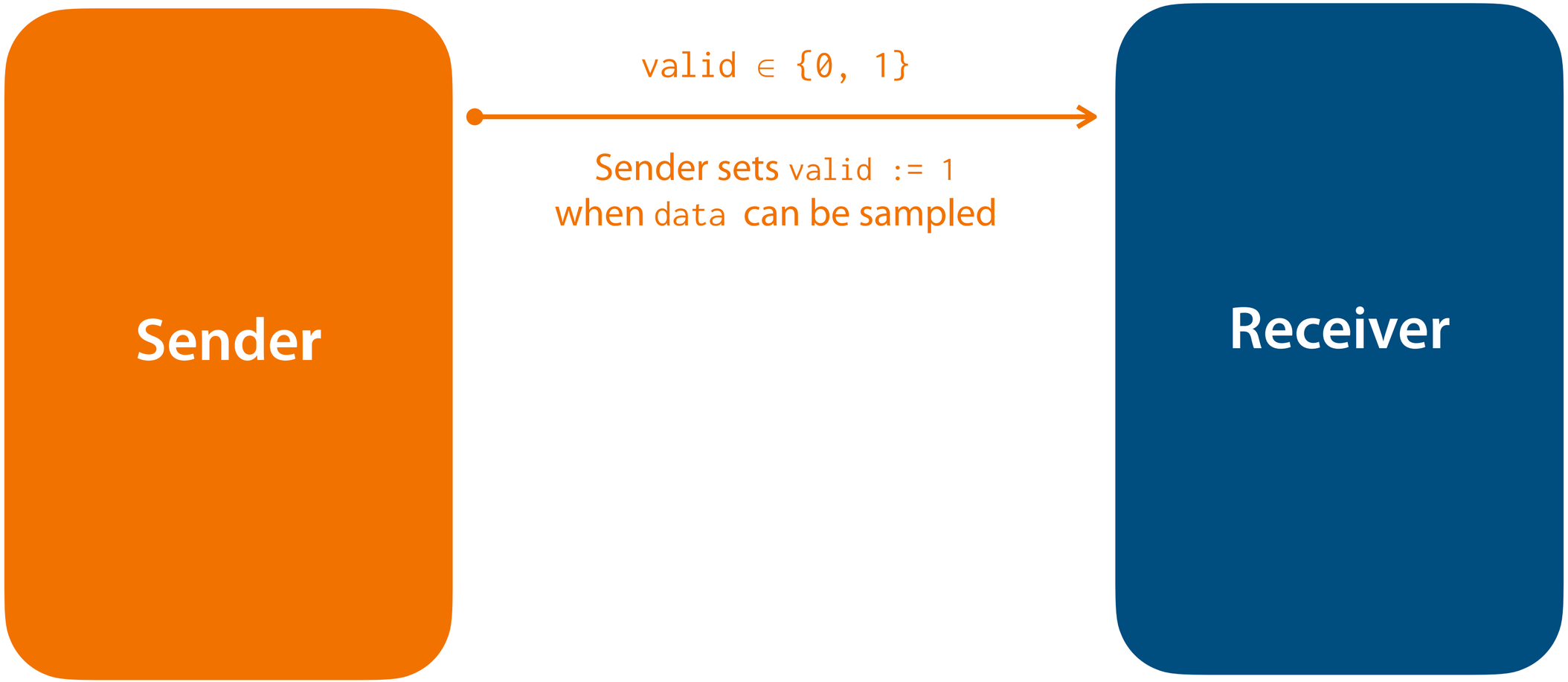
Running example: Ready-Valid Handshake

Ready-Valid Handshake

Mediates data transfer between a sender & a receiver

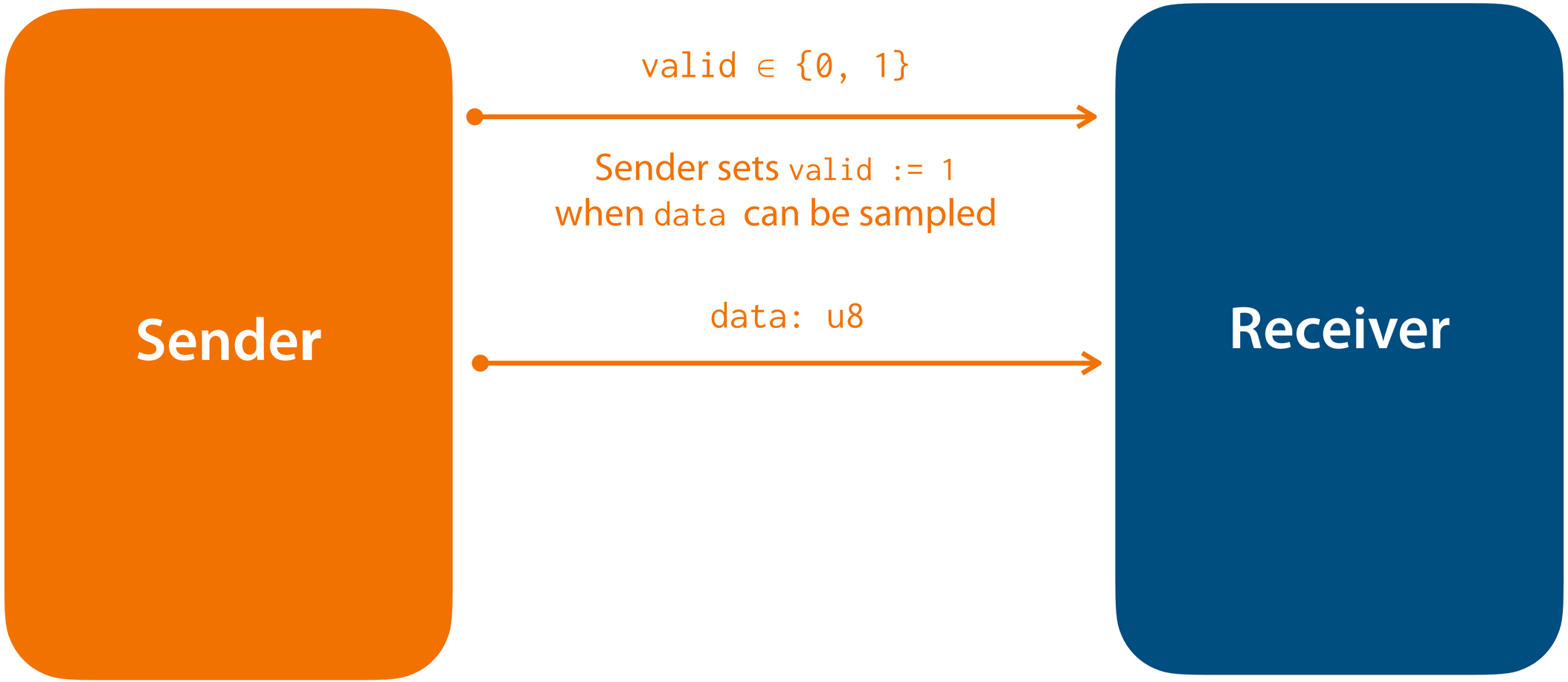
Ready-Valid Handshake

Mediates data transfer between a sender & a receiver



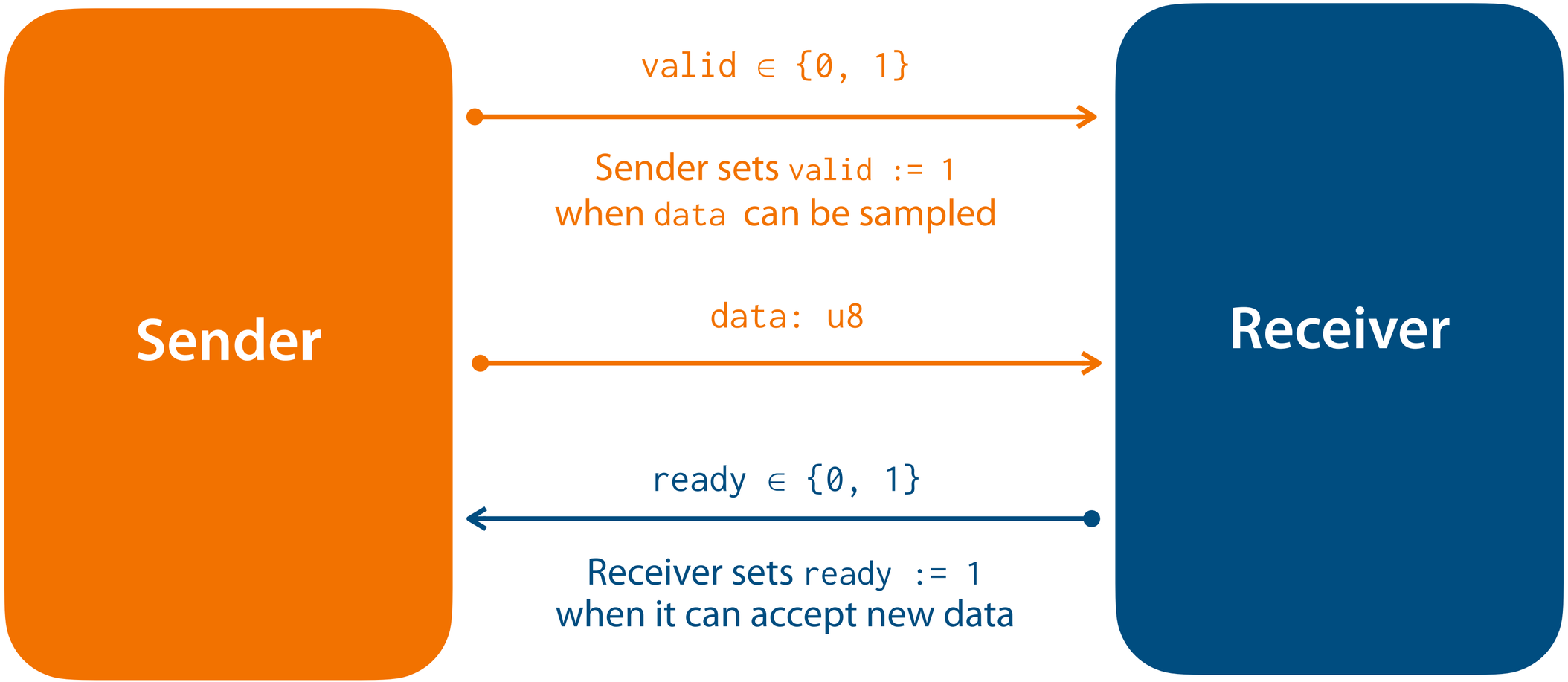
Ready-Valid Handshake

Mediates data transfer between a sender & a receiver



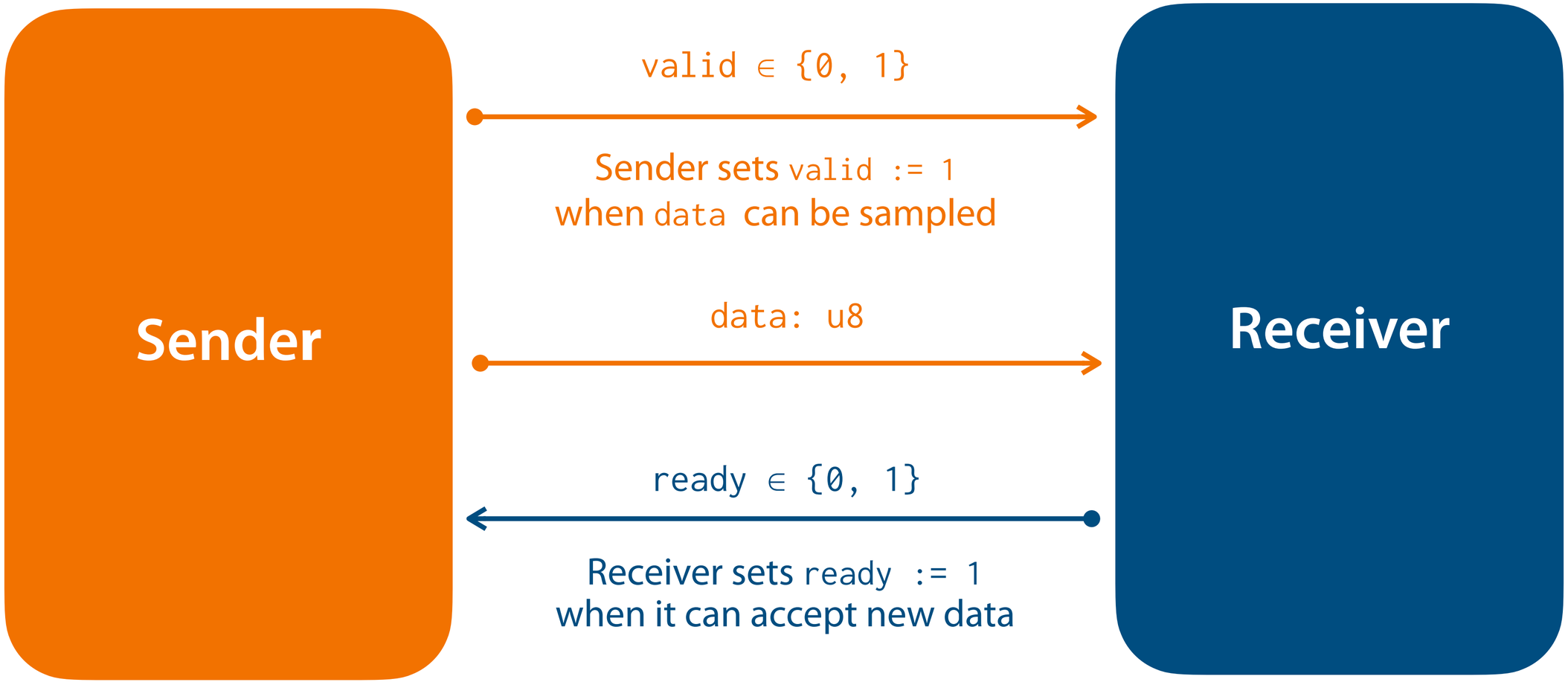
Ready-Valid Handshake

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:

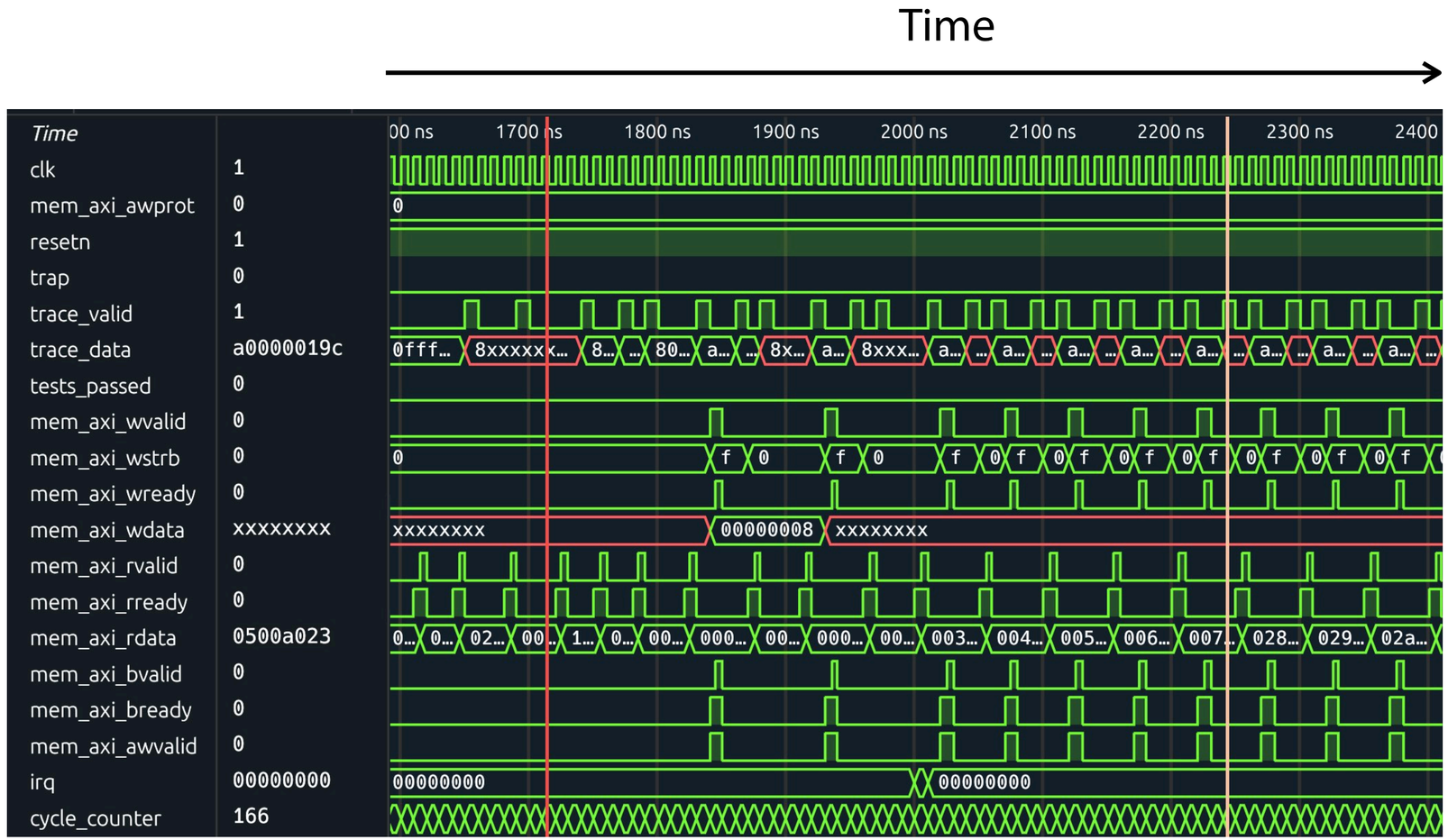
These protocols are often stated implicitly (in English), making testing & debugging hard!

Problem:

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

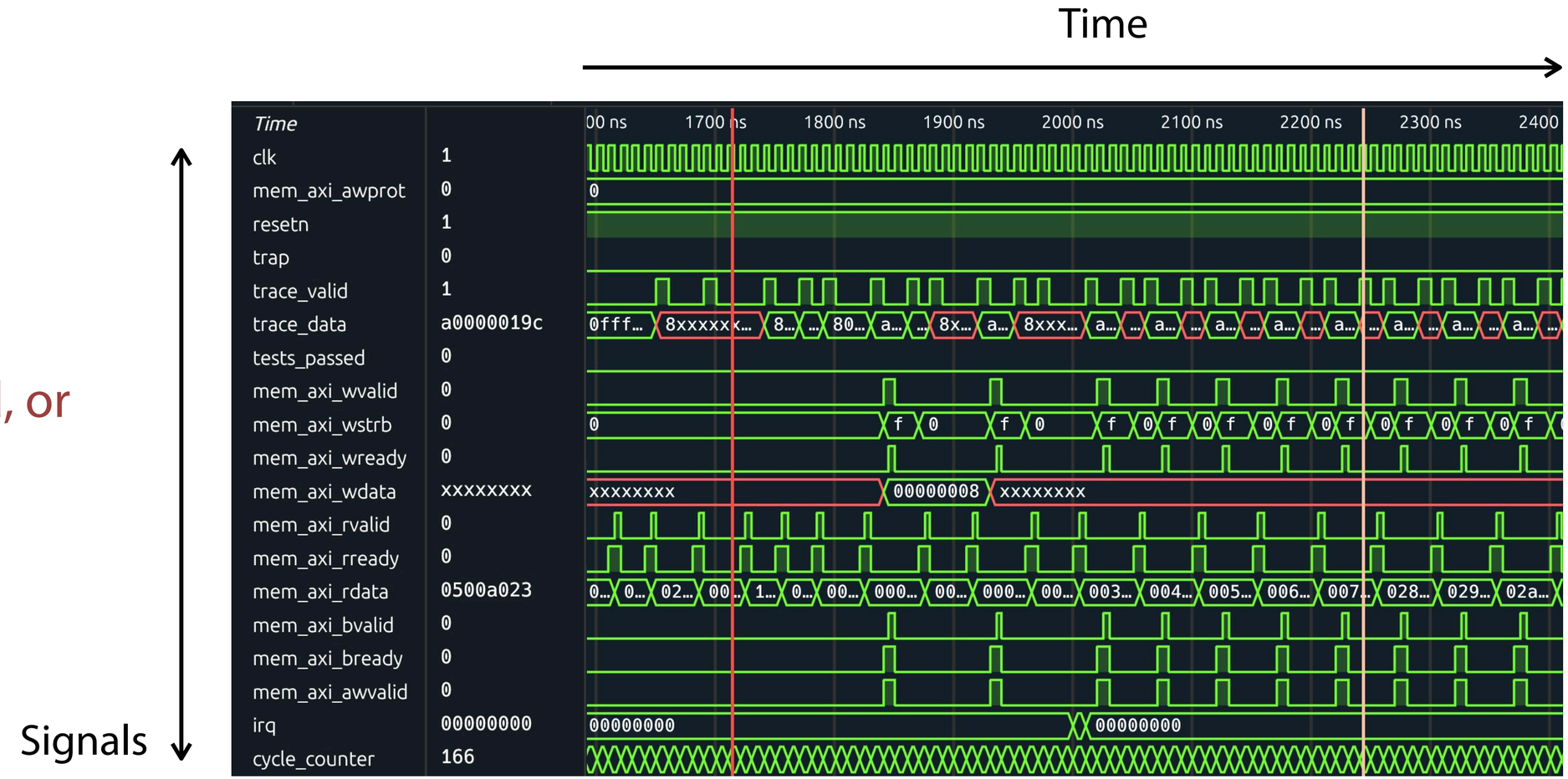
Debugging hardware is hard

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



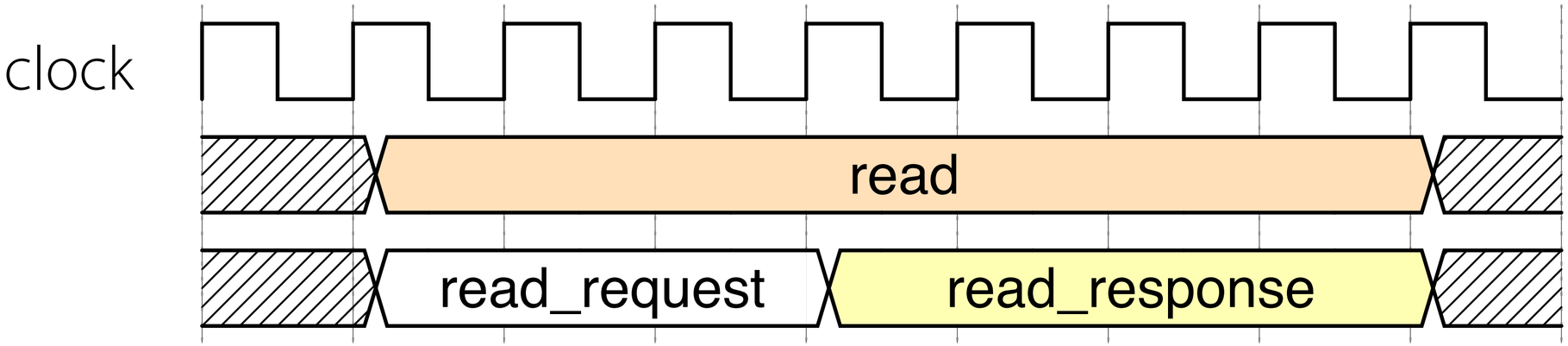
Debugging hardware is hard

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

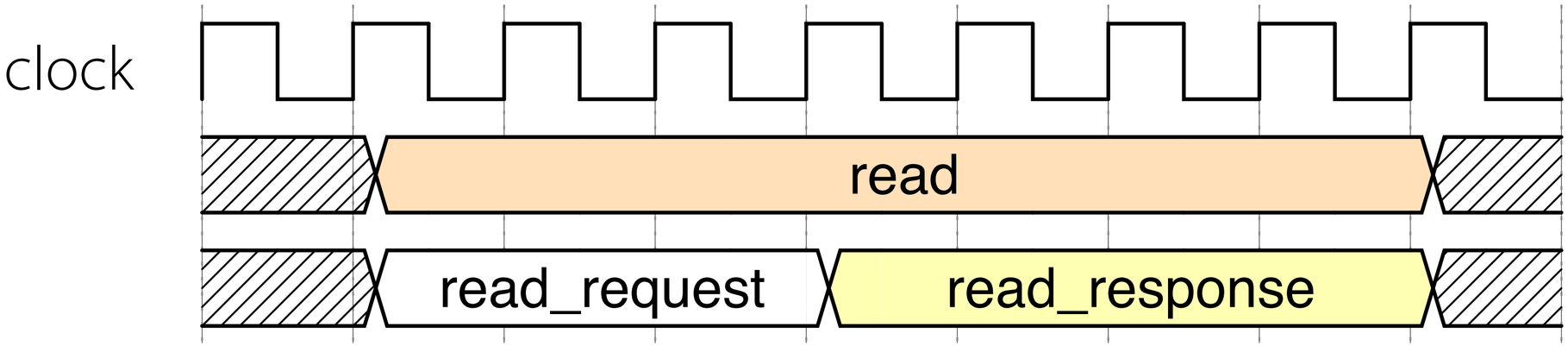


What if we had waveform viewers that looked like this instead?

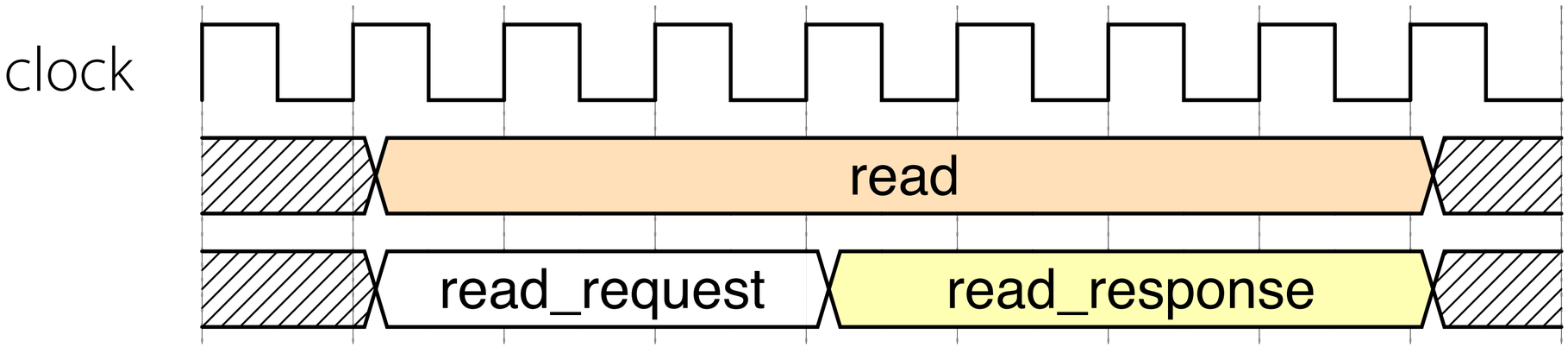
What if we had waveform viewers that looked like this instead?



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.

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

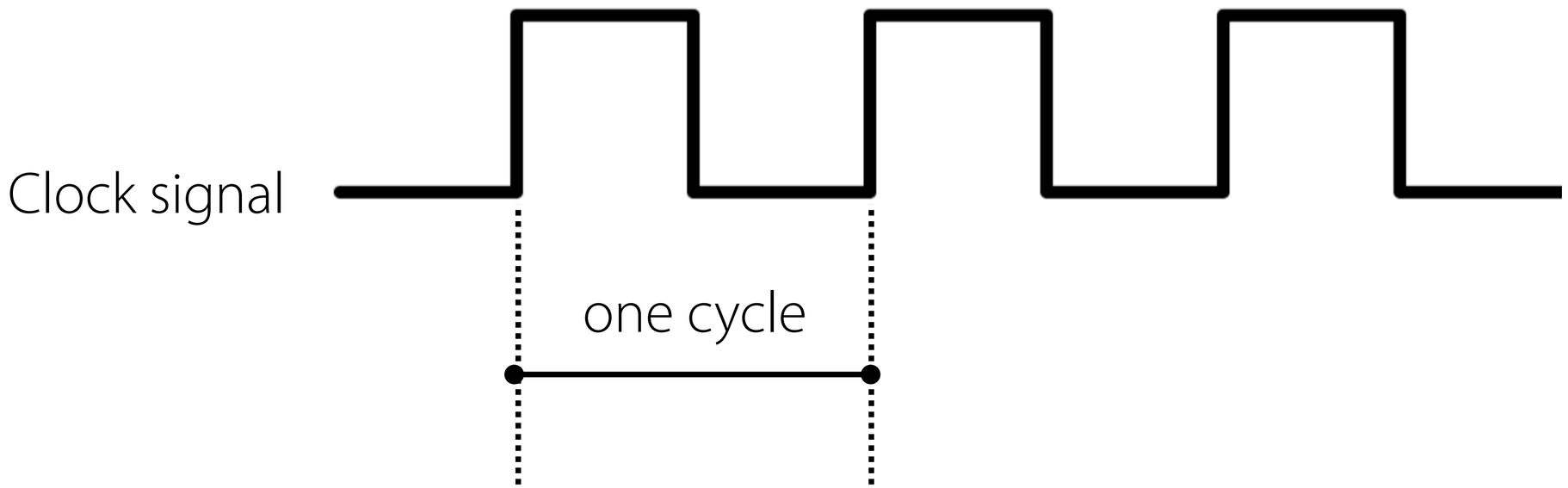
```
prot send_data<DUT: ReadyValid>(data: u8) {
    DUT.valid := 1;
    DUT.data := data;
    while (DUT.ready ≠ 1) {
      step();
    }

}
```

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

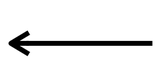
Aside: Paso's `step()` primitive

`step()` advances the clock by one cycle



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

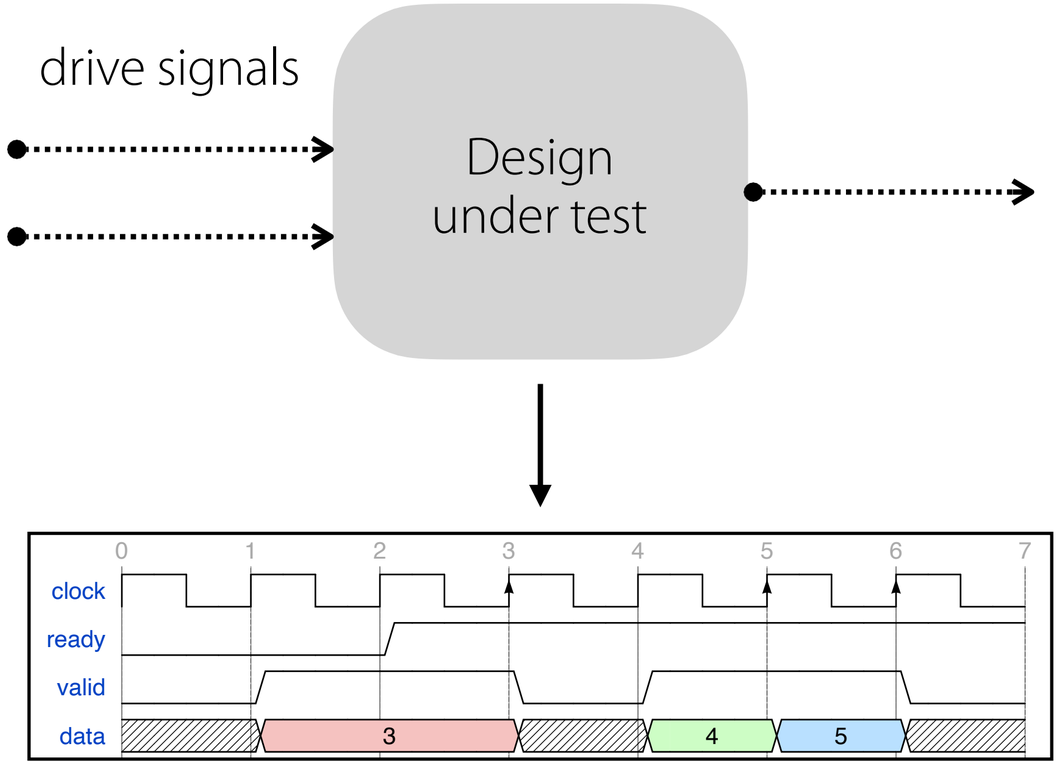
Simple imperative semantics, just like software!

```
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?

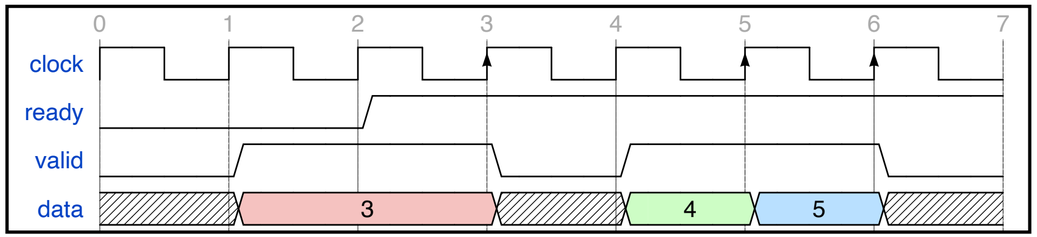
What can we do with one single protocol?

1. Run concrete tests (drive hardware modules)

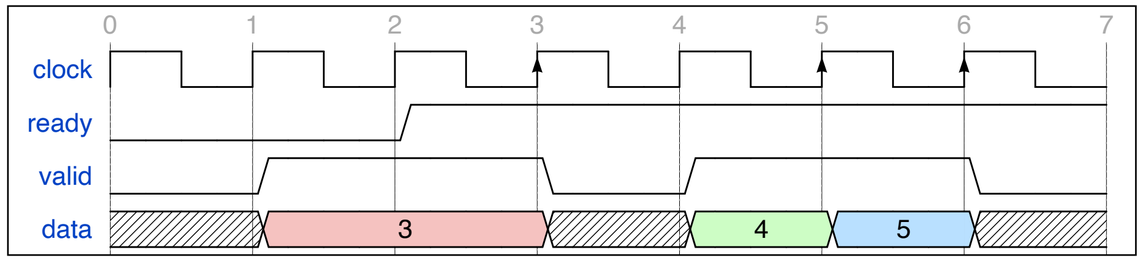


What can we do with one single protocol?

1. Run concrete tests (drive hardware modules)



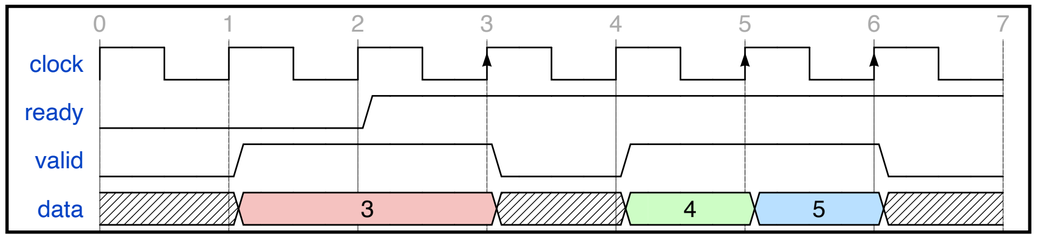
2. Infer transactions from waveforms



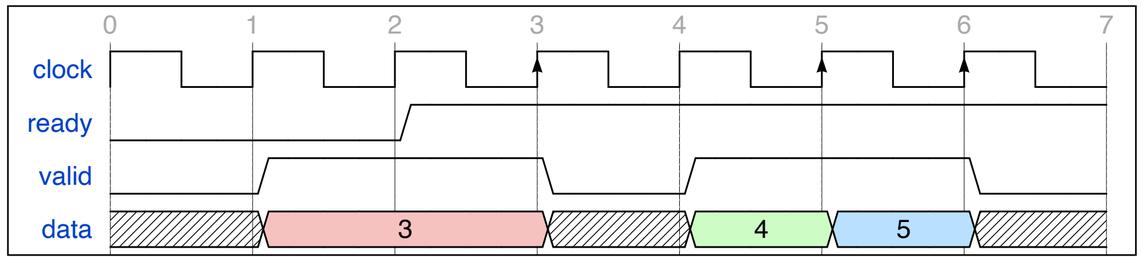

```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

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

Hardware module implementation

Transactions to execute

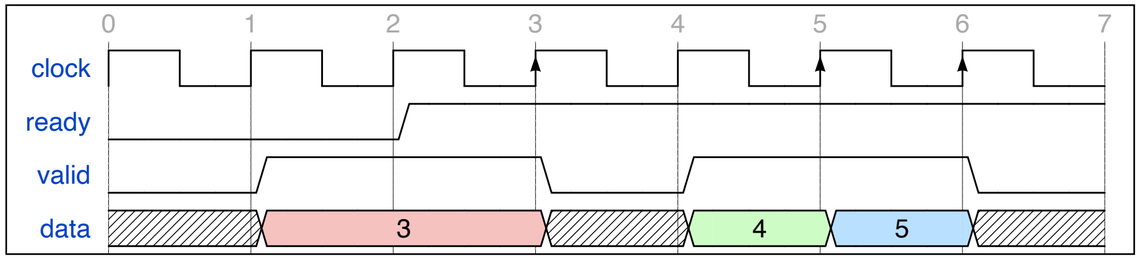
```
send_data(3);
send_data(4);
send_data(5);
```

Paso protocol spec

SystemVerilog

Interpreter

Waveform

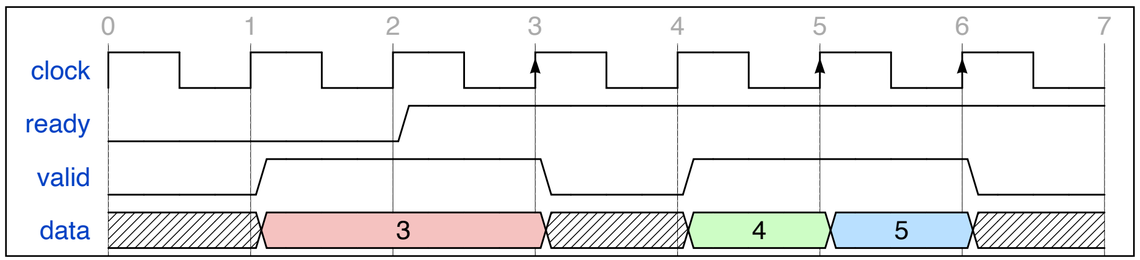


Reconstructor

(the cool part!)

Paso protocol spec

Waveform



Reconstructor

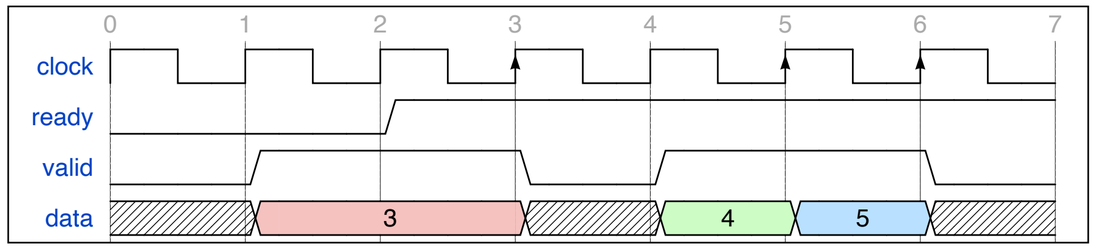
Transaction trace

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

Applying the Reconstructor on the Ready-Valid Example

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

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



Reconstructor

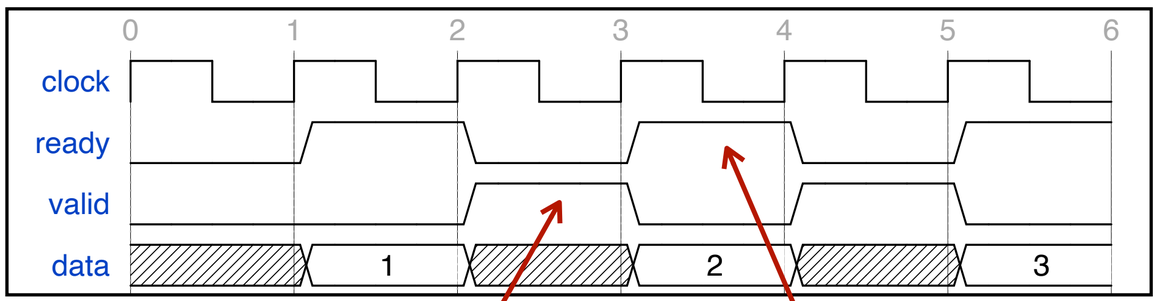
```
send_data(3); // cycle 1-3
send_data(4); // cycle 4-5
send_data(5); // cycle 5-6
```

Abstraction level raised from signals to transactions!

Applying the Reconstructor on the Ready-Valid Example

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

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

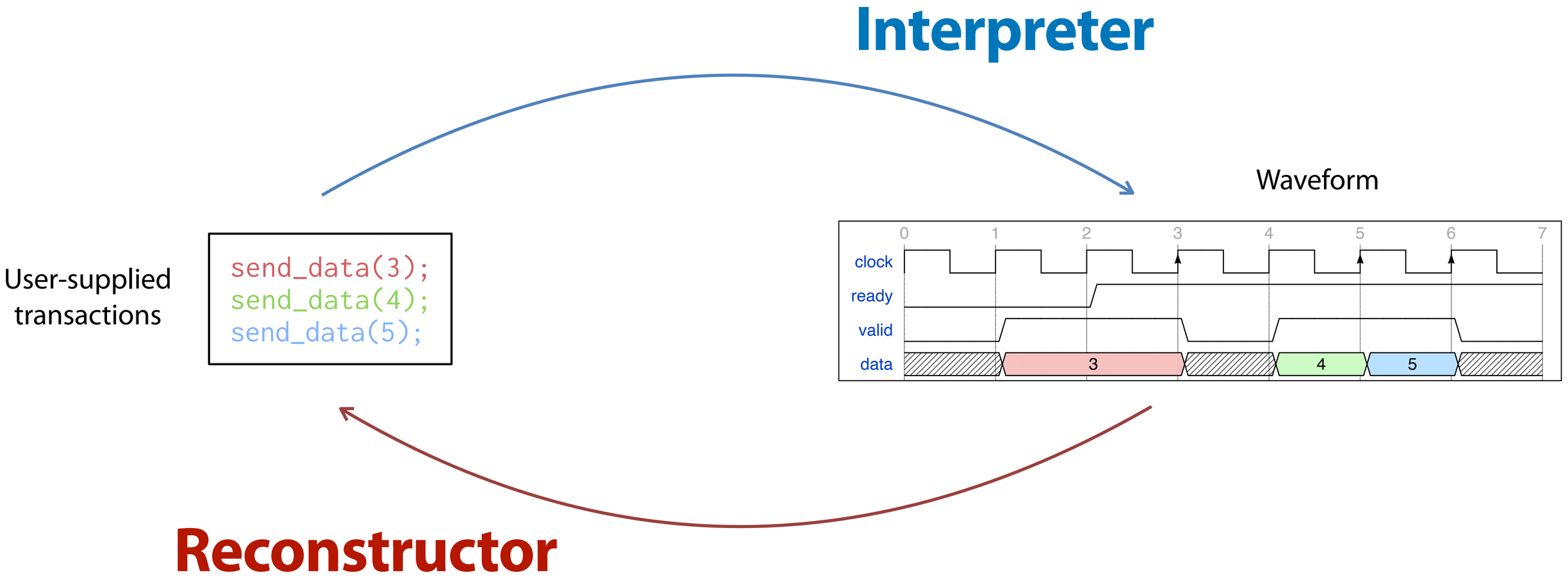


Reconstructor

```
Error: no matching
transactions
```

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

Round-trip property



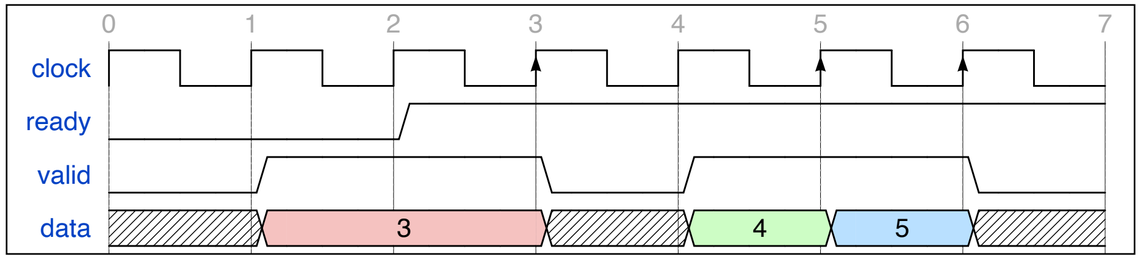
Round-trip property

Interpreter

User-supplied transactions

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

Waveform



Reconstructor

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

How does the reconstructor work?

Assignments as Constraints

`foo := 0`

means

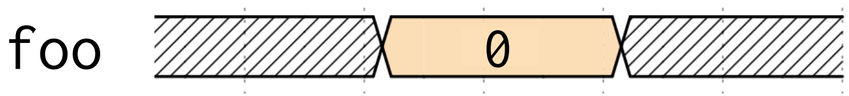
"Check if the current waveform value of `foo` is equal to `0`"

Assignments as Constraints

`foo := 0`

means

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



Matches!

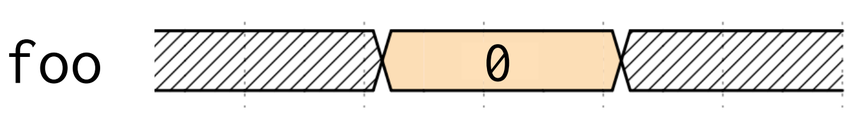
✓

Assignments as Constraints

`foo := 0`

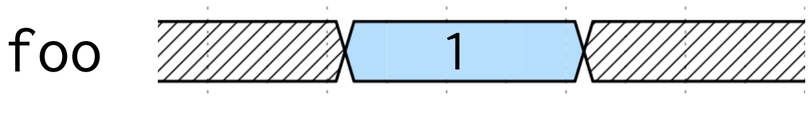
means

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



Matches!

✓



Inconsistent!

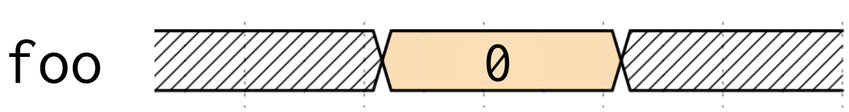
X

Assignments as Constraints

`foo := 0`

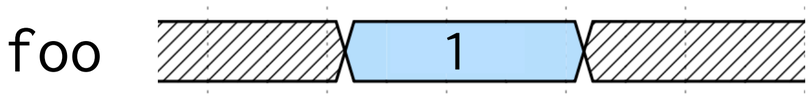
means

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



Matches!

✓



Inconsistent!

X

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

Assignment in protocol body

LHS := RHS

Constraint in reconstructor

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)`

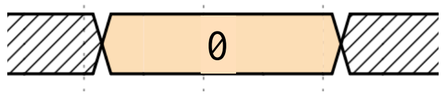
`foo := a` ⟶ `a = waveform(foo)`

Assignment in protocol body

Constraint in reconstructor

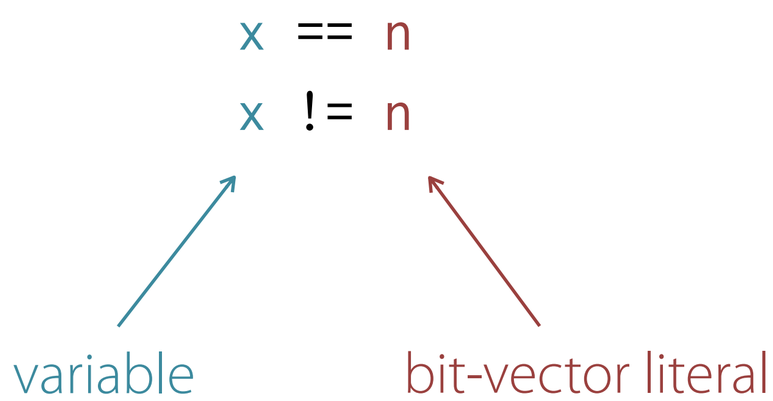
`LHS := RHS` ⟶ `RHS` = waveform value of **LHS** at the current clock cycle

`foo := 0` ⟶ `0 = 0`

`foo` 

`foo := a` ⟶ `a = 0`

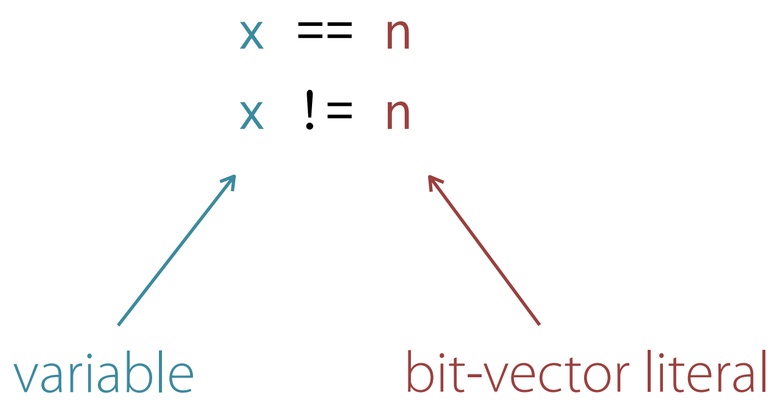
No SMT solvers needed!



No SMT solvers needed!

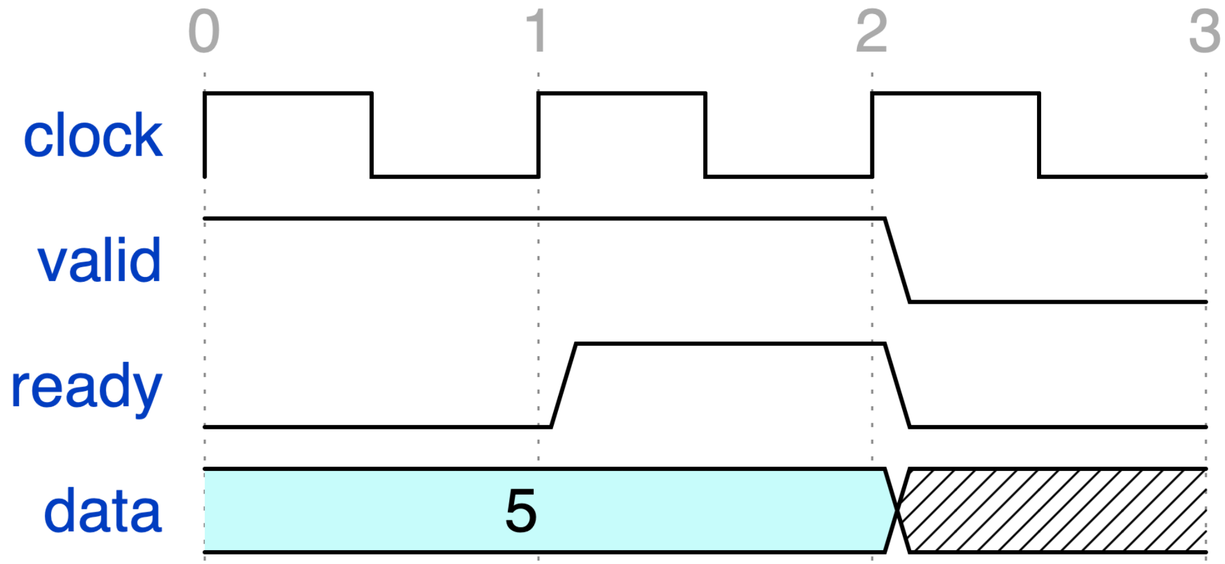
All constraints in our DSL are equality constraints where one side is a constant

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



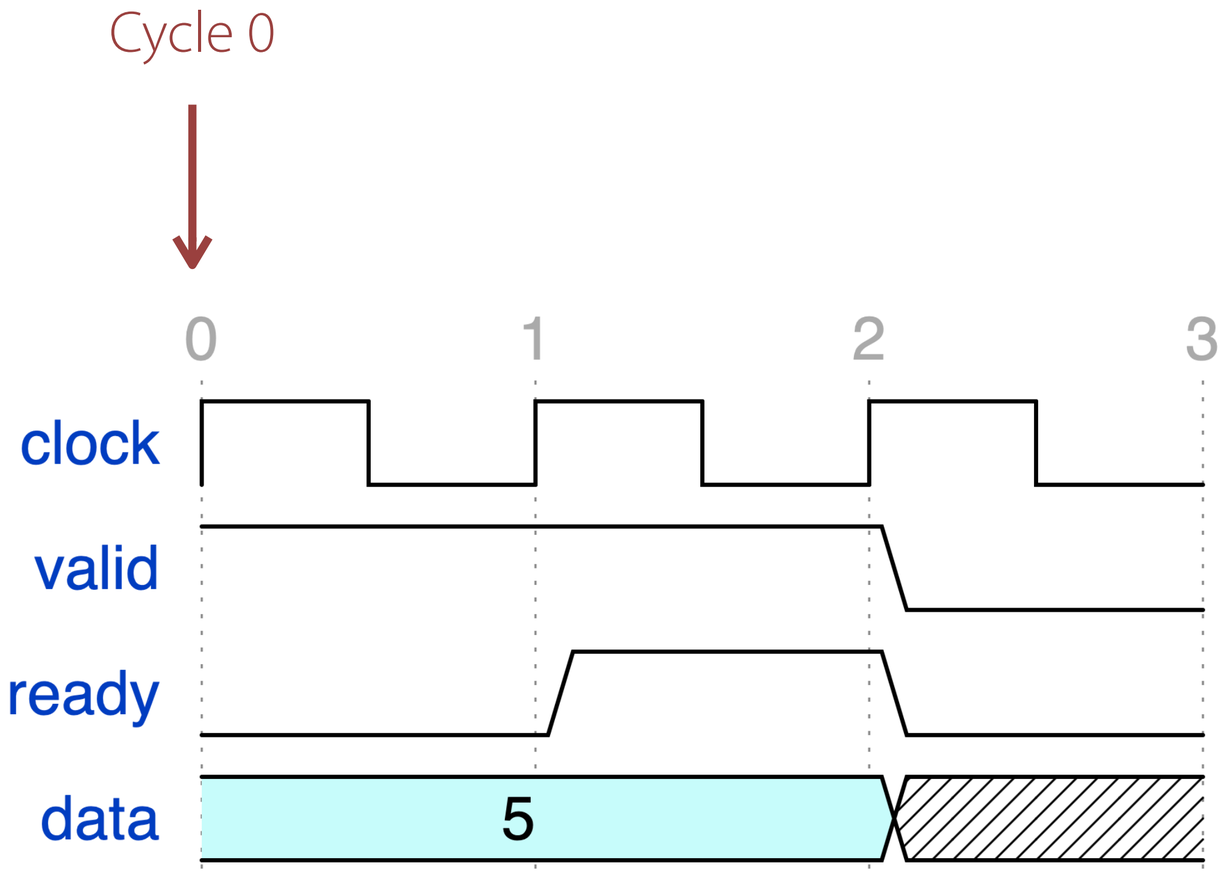
Idea: symbolically execute protocols & compare against waveform

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



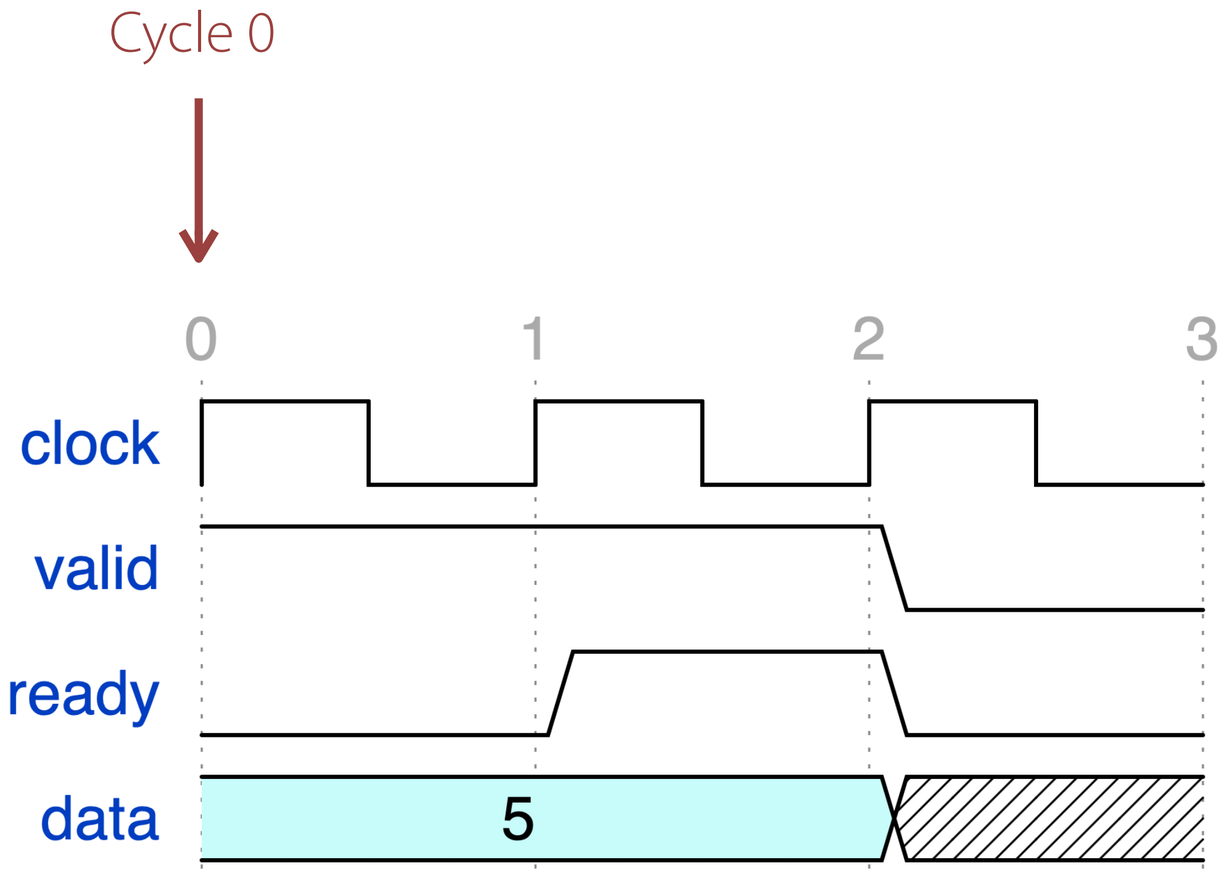
Constraints: ∅

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



Constraints: ∅

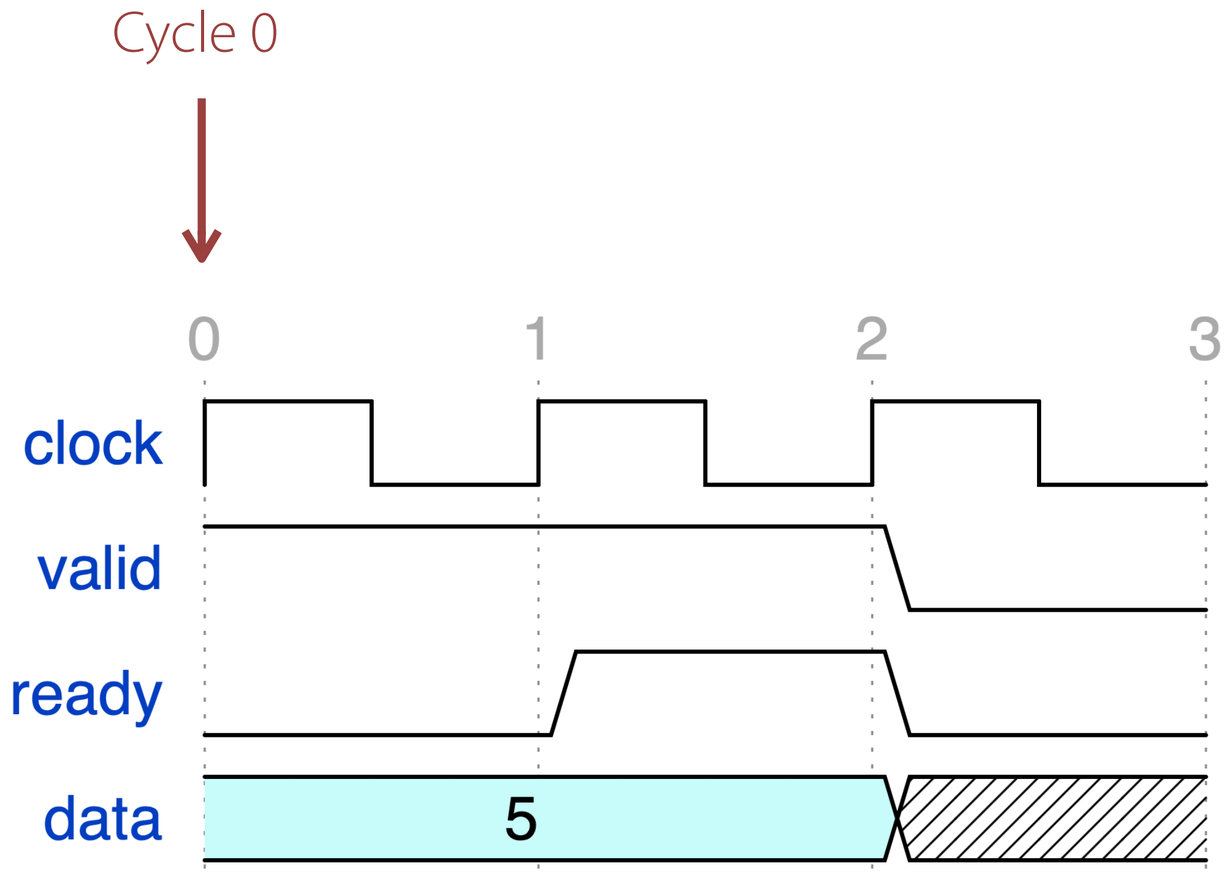
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```



Constraints: ∅

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

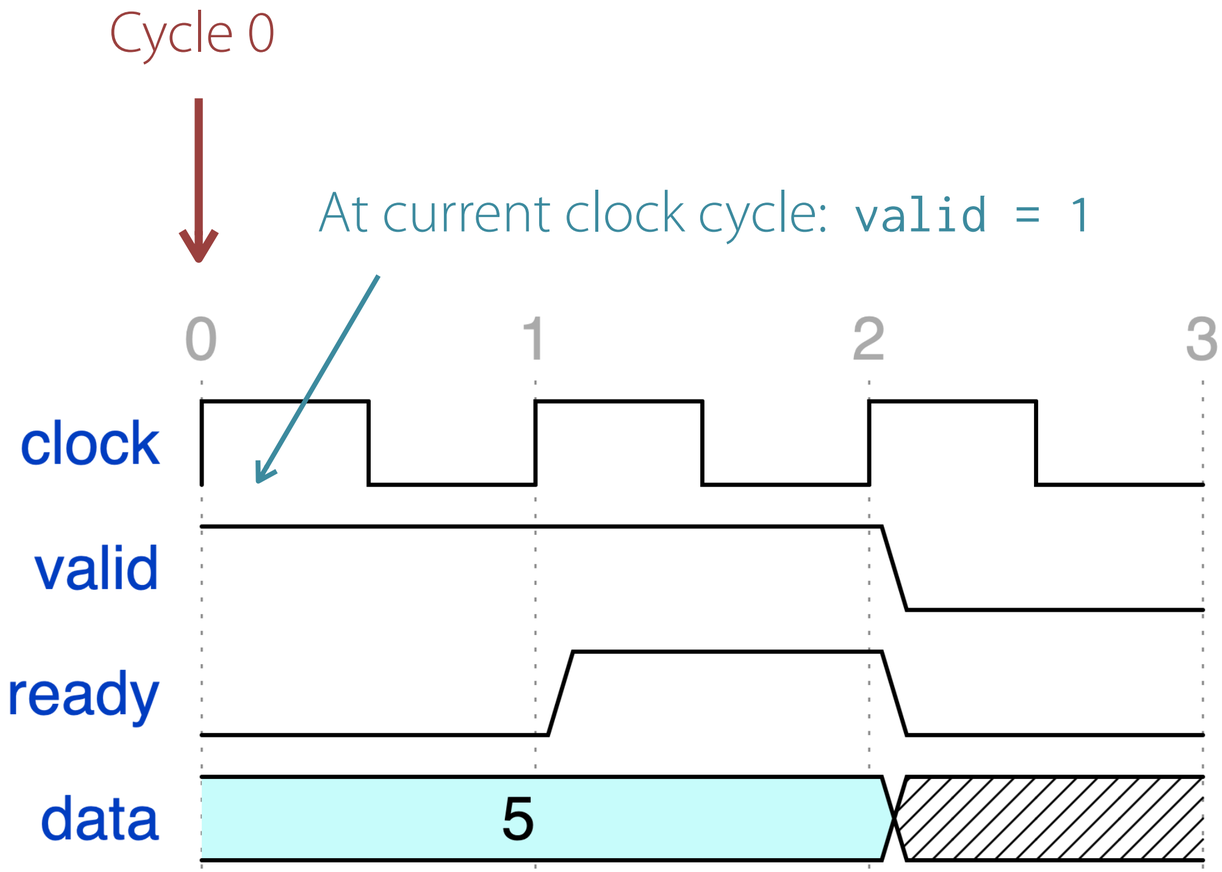
Check if current waveform value of `valid =? 1`



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`



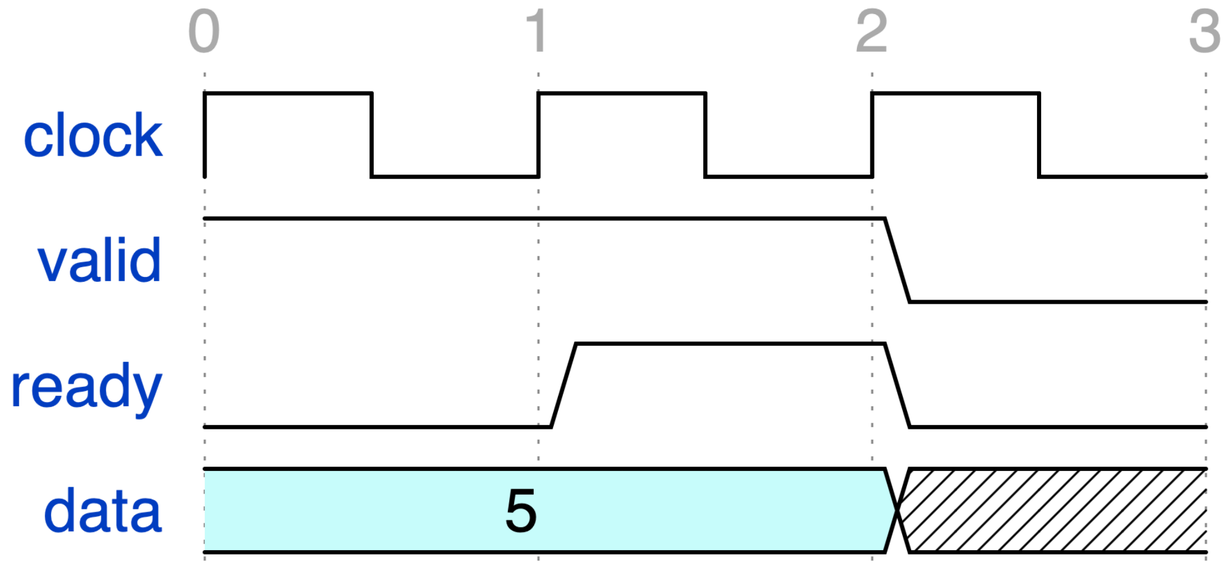
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
    DUT.valid := 1;
    DUT.data  := data;
    while (DUT.ready ≠ 1) {
      step();
    }
    step();
}
```

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

Constraints: `{ 1 = 1 }`

Cycle 0

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



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

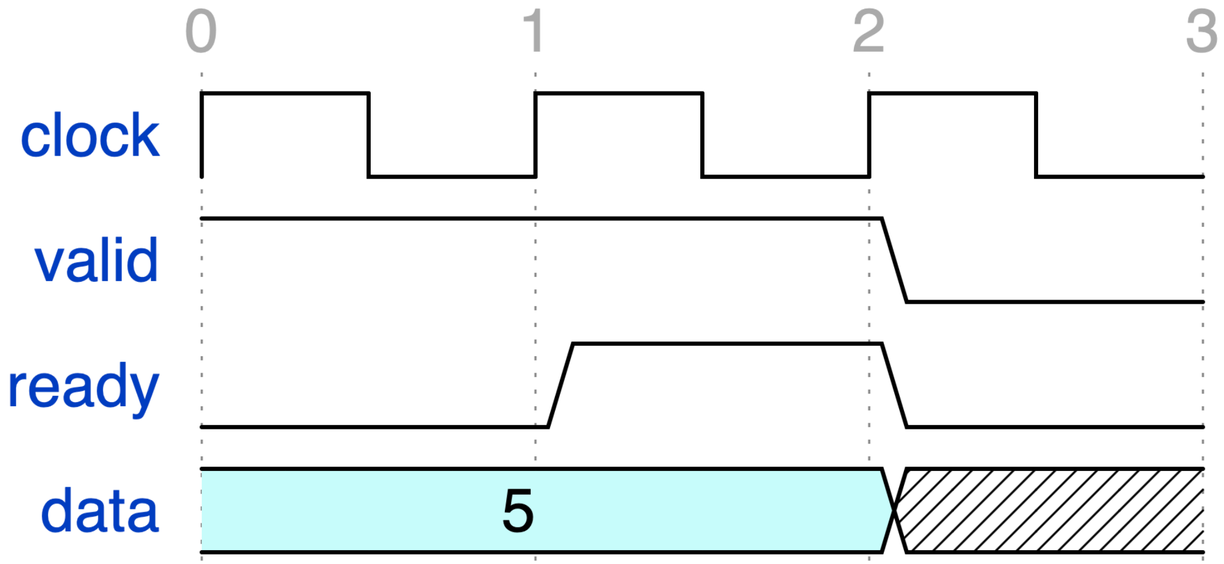
Check if current waveform value of `valid =? 1`

Constraints: `{ 1 = 1 }`

Cycle 0

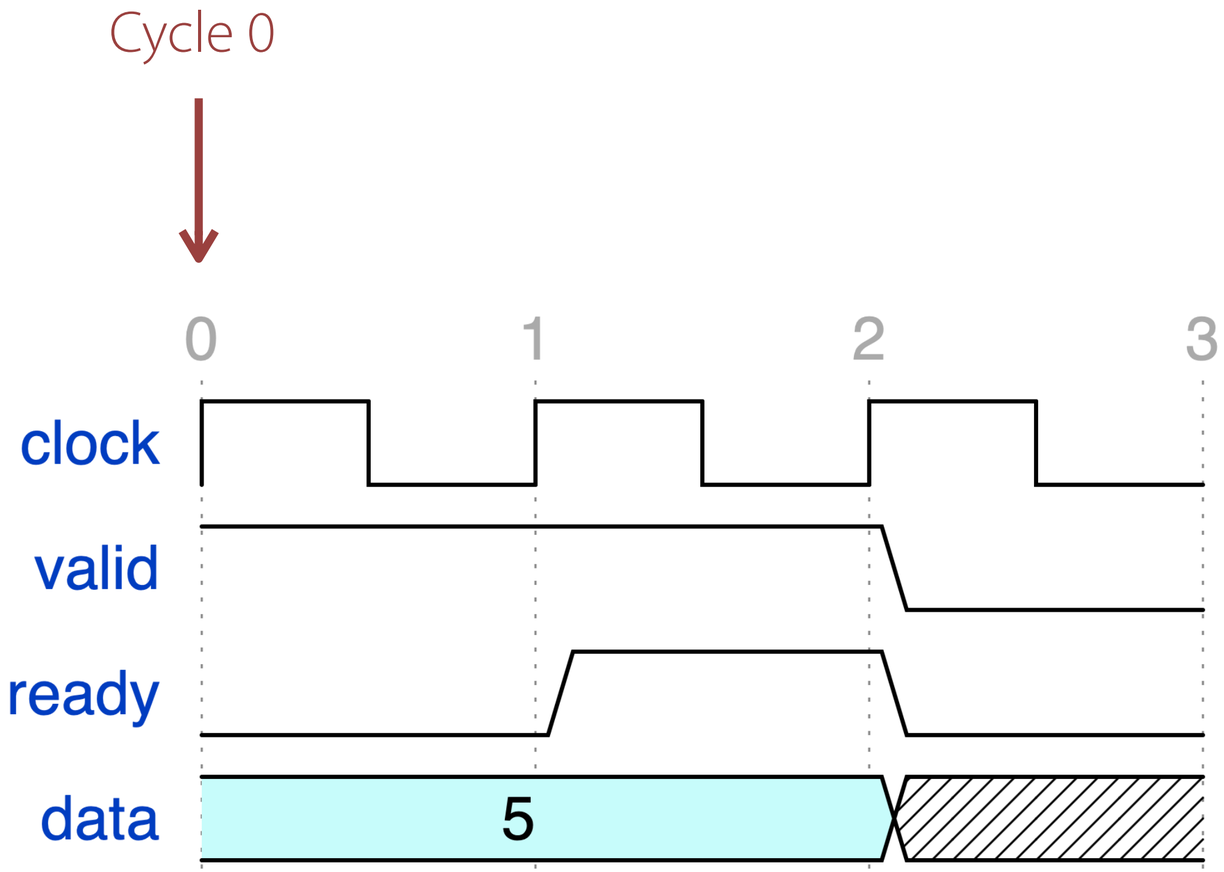
RHS of the stmt
`DUT.valid := 1`

Waveform value of `valid` @ current clock cycle



Constraints: { 1 = 1 }

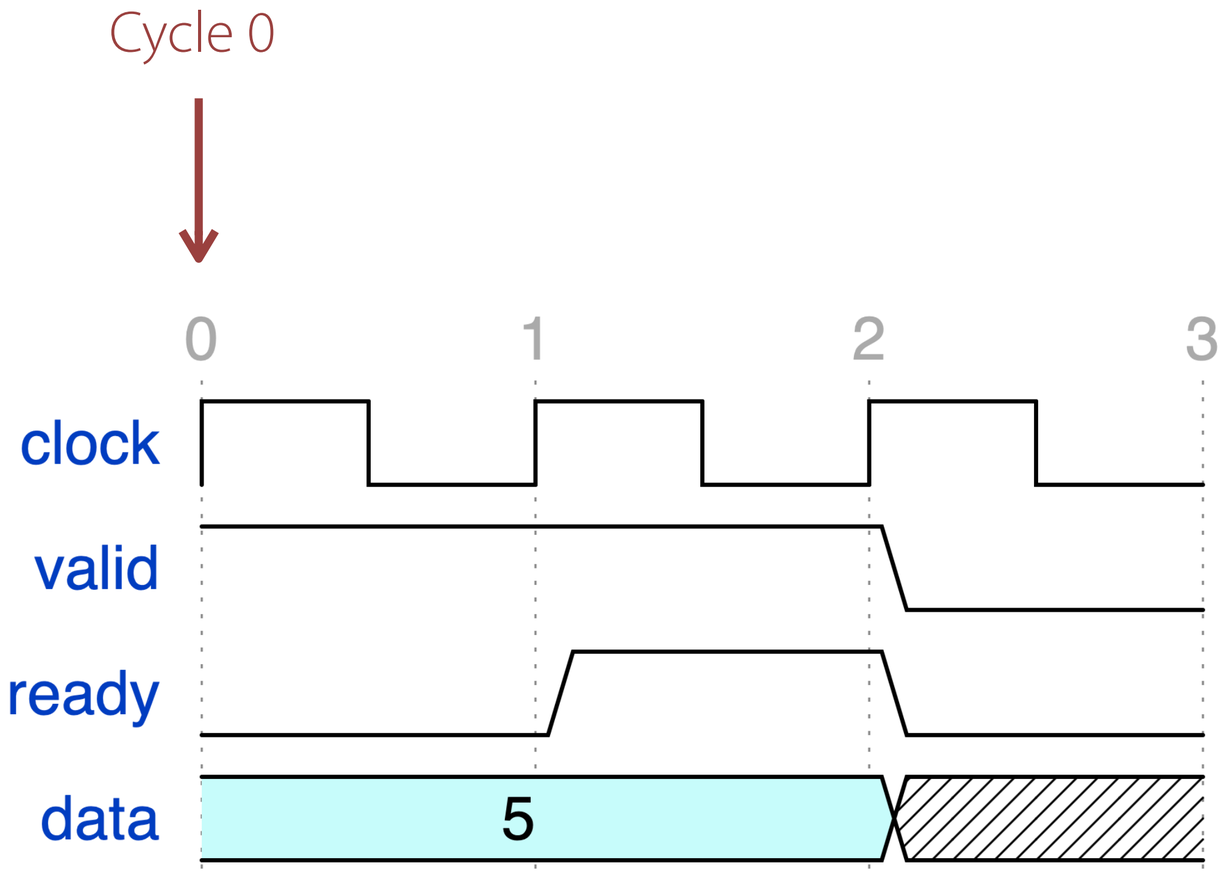
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```



Constraints: { 1 = 1 }

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

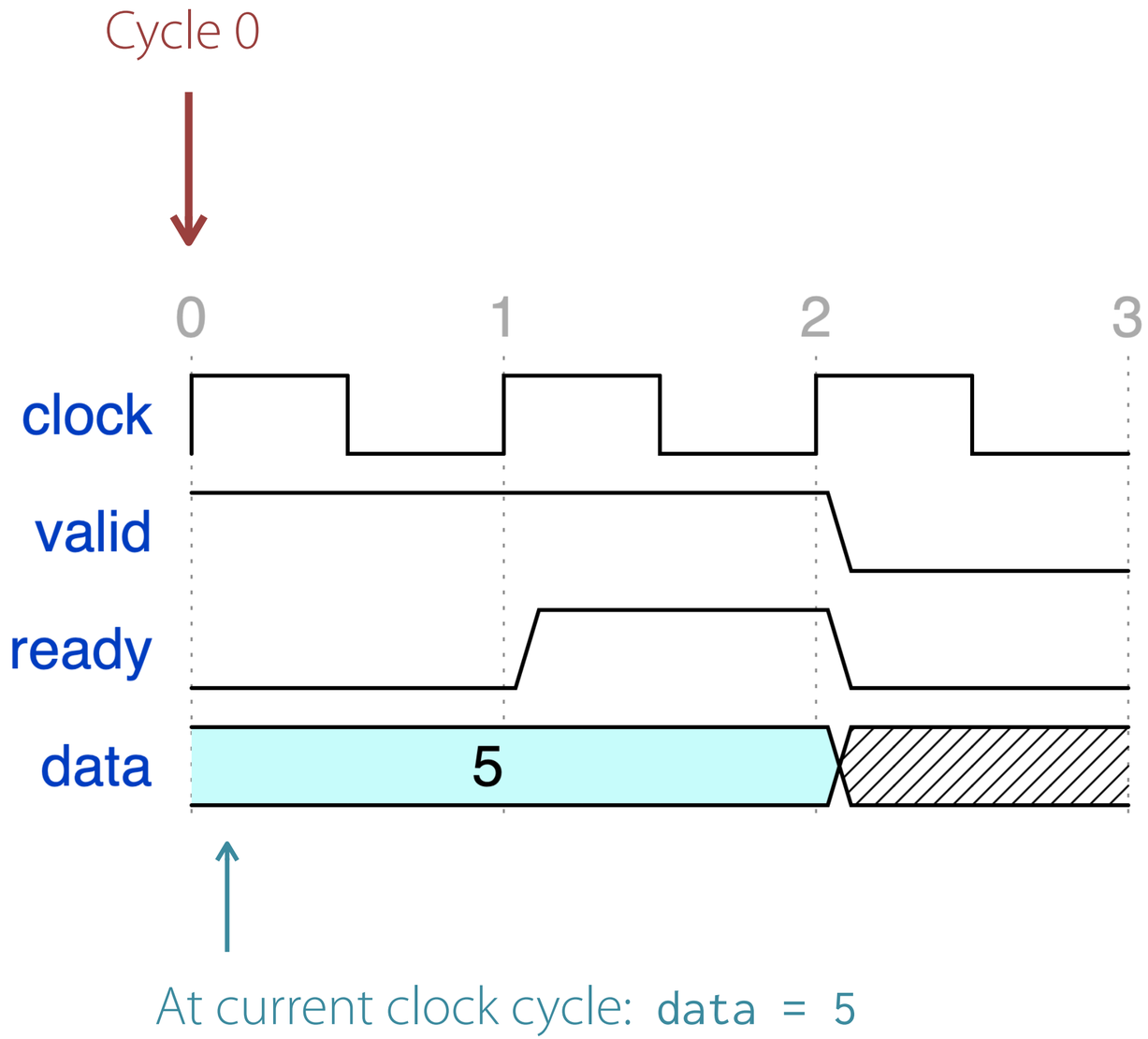
Look up current value of RHS (`data`) from waveform



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

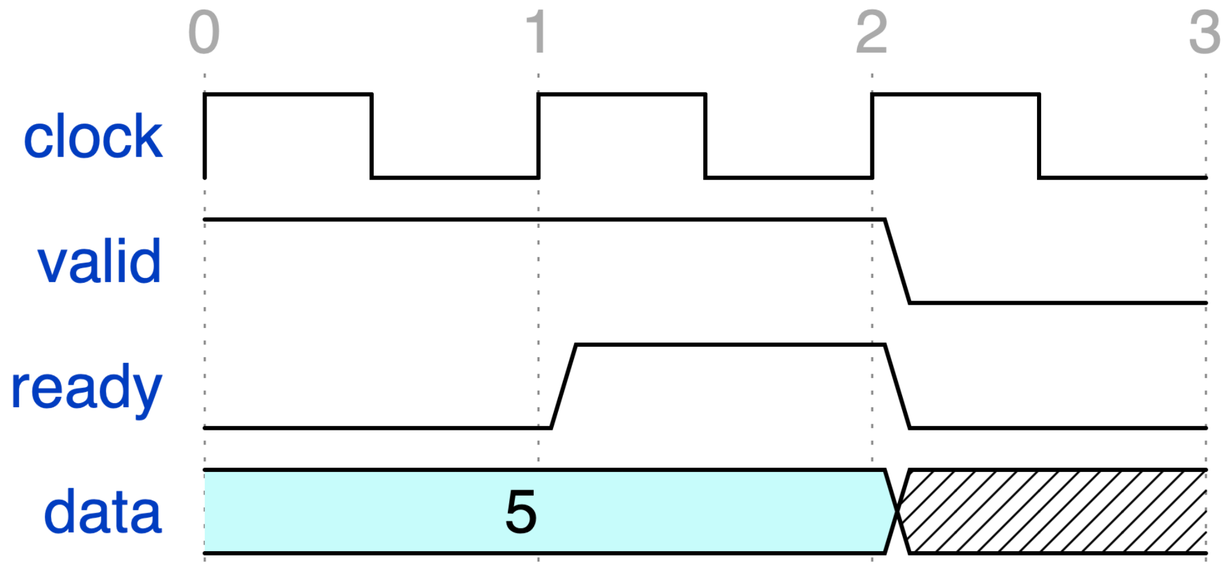


```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

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

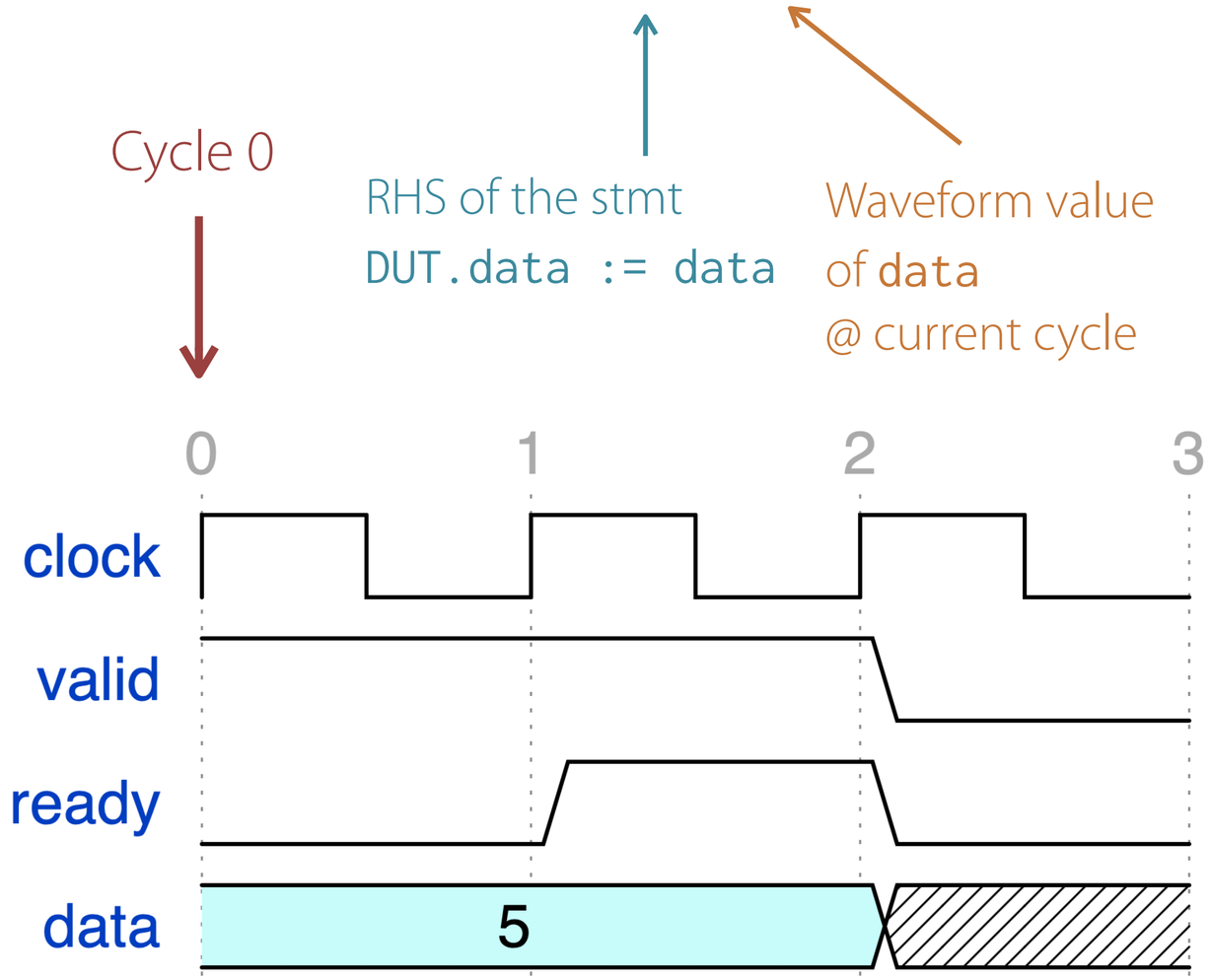
Add a new constraint `data = 5`

Cycle 0



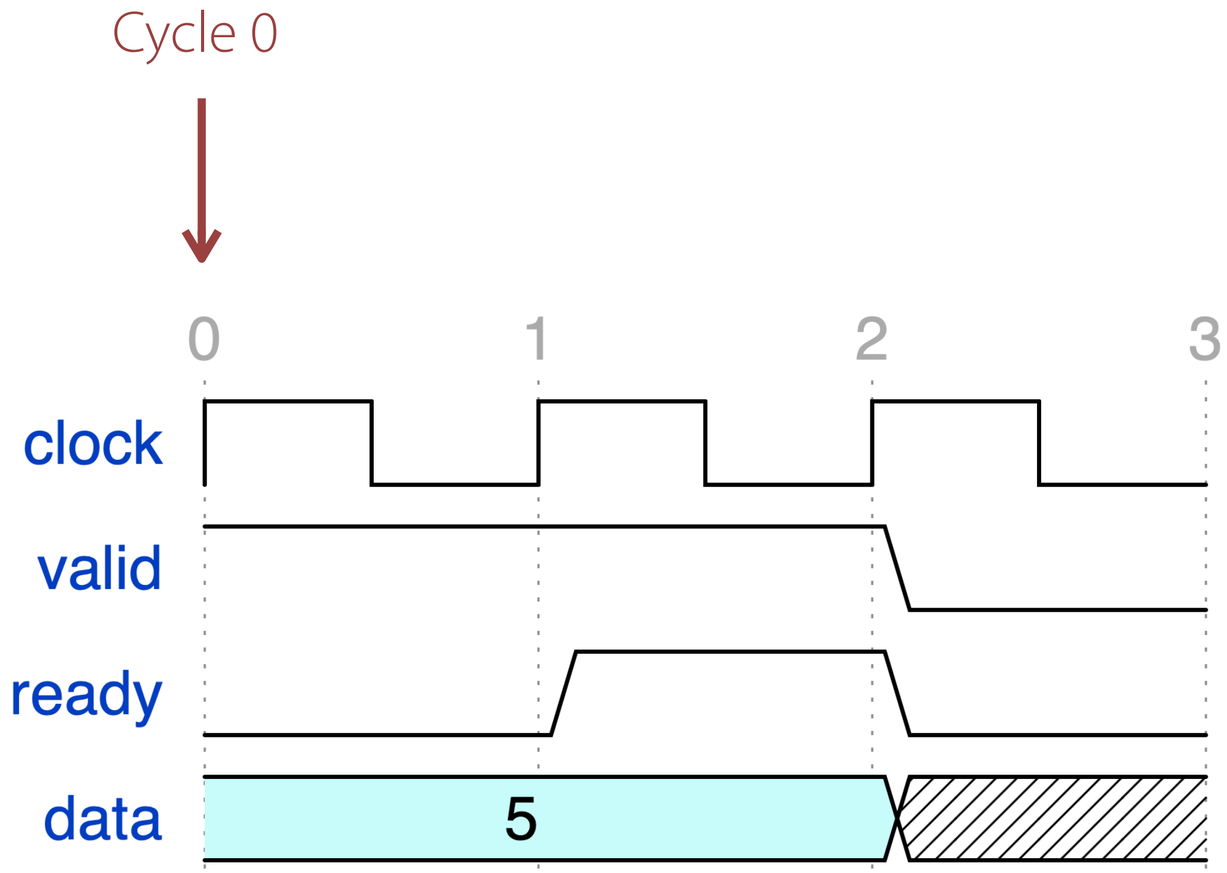
```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

Constraints: { 1 = 1, data = 5 }



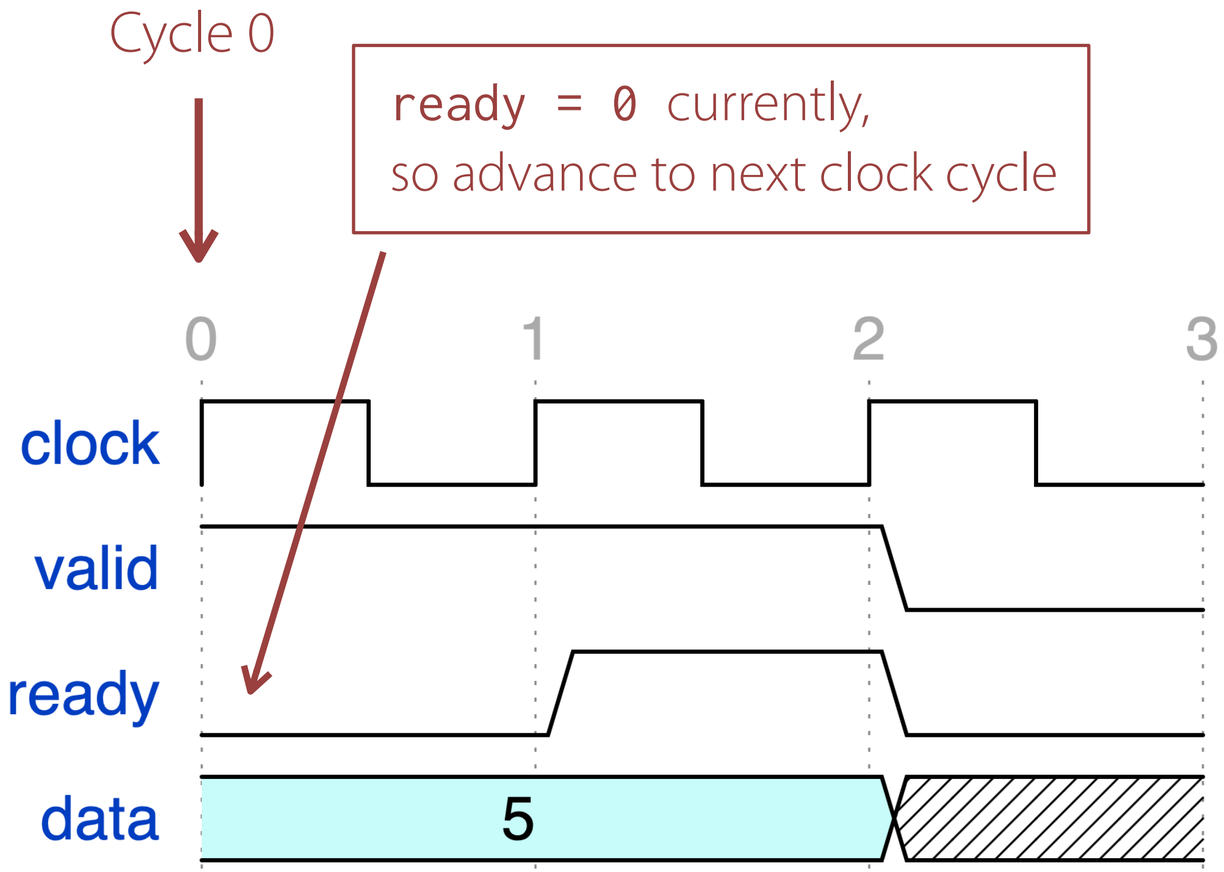
Constraints: `{ 1 = 1, data = 5 }`

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



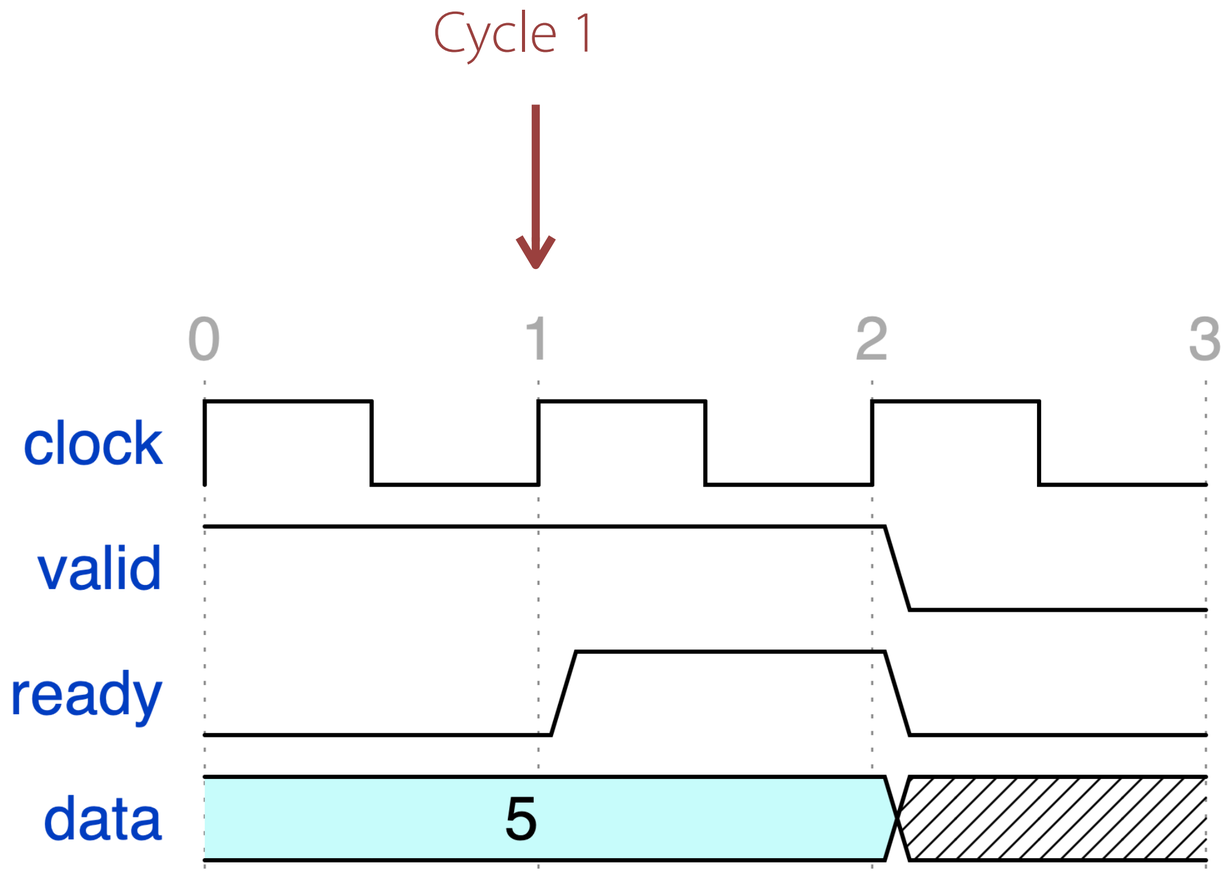
Constraints: `{ 1 = 1, data = 5 }`

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



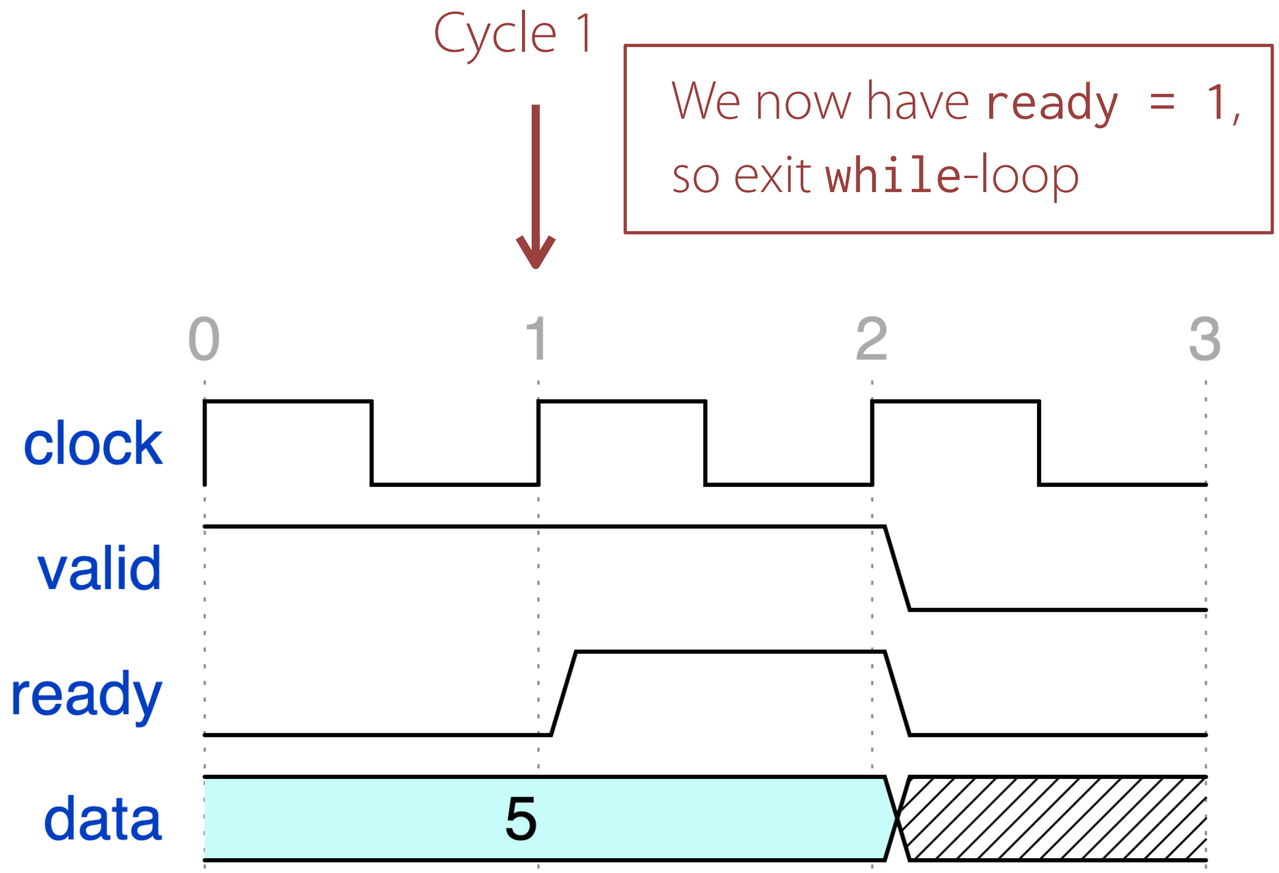
Constraints: `{ 1 = 1, data = 5 }`

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



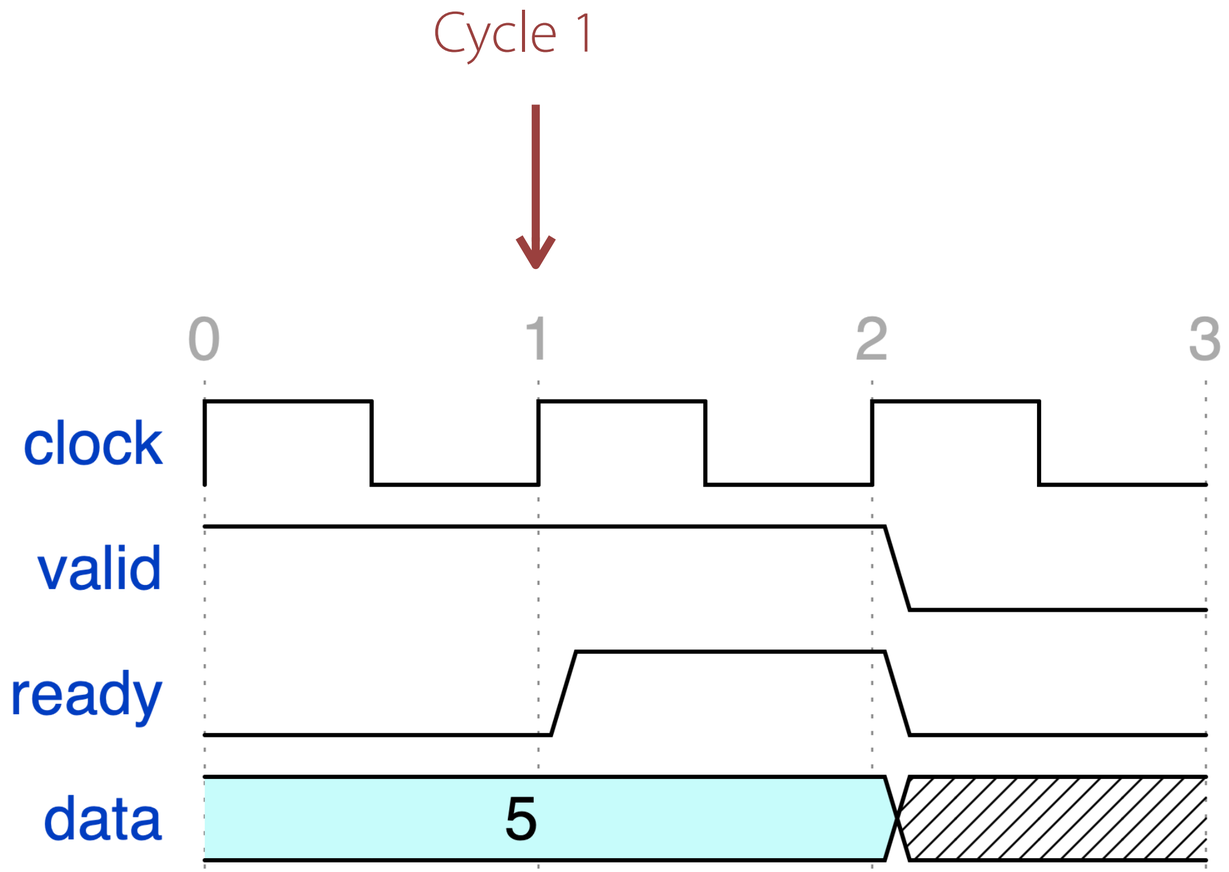
Constraints: `{ 1 = 1, data = 5 }`

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



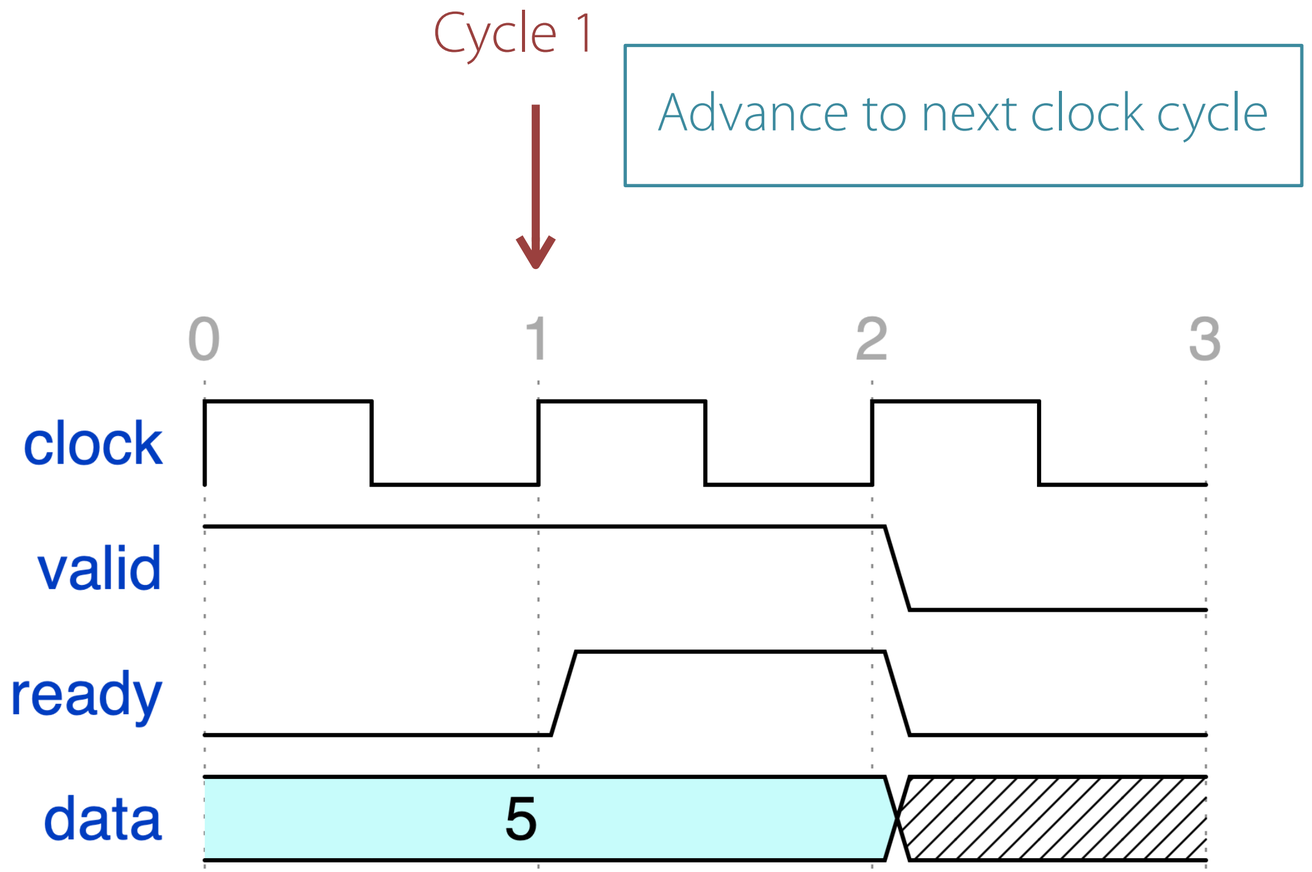
Constraints: `{ 1 = 1, data = 5 }`

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



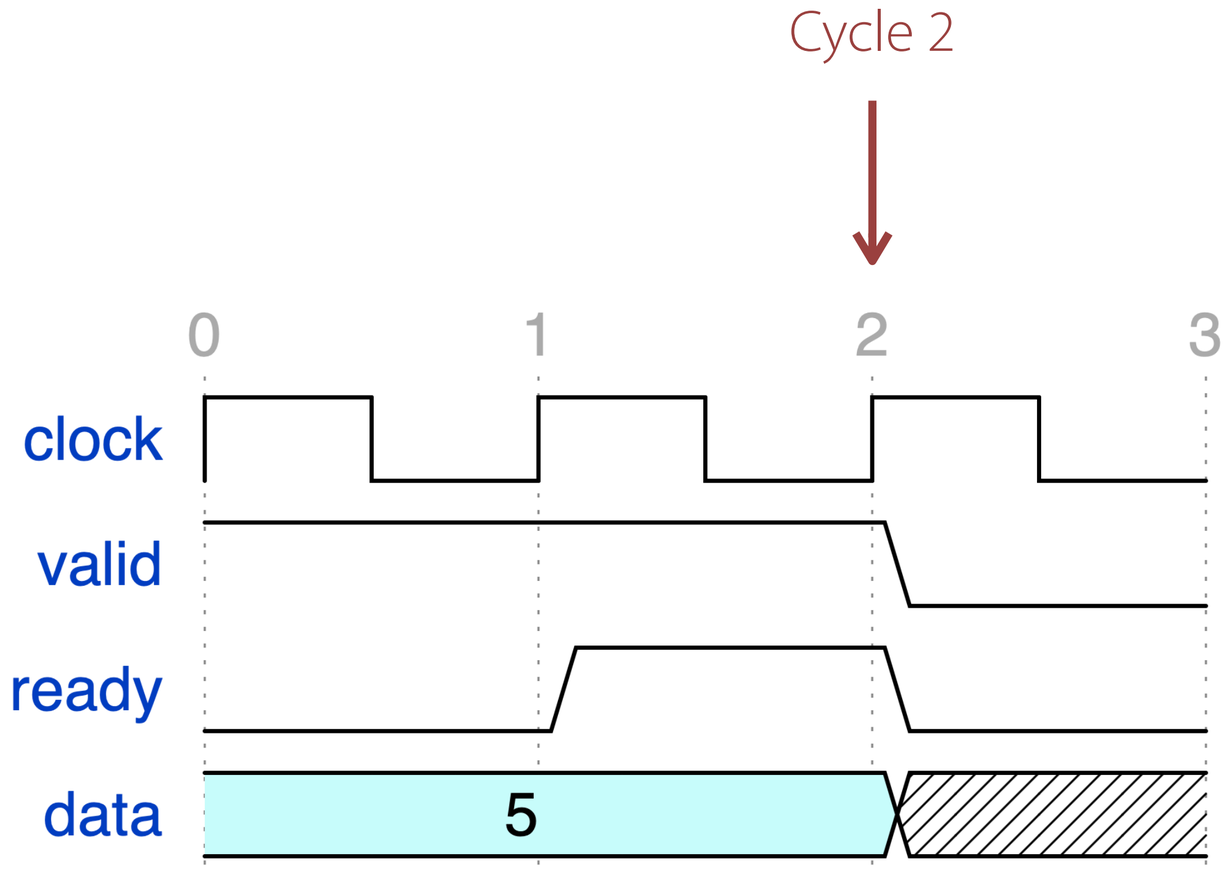
Constraints: `{ 1 = 1, data = 5 }`

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



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

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

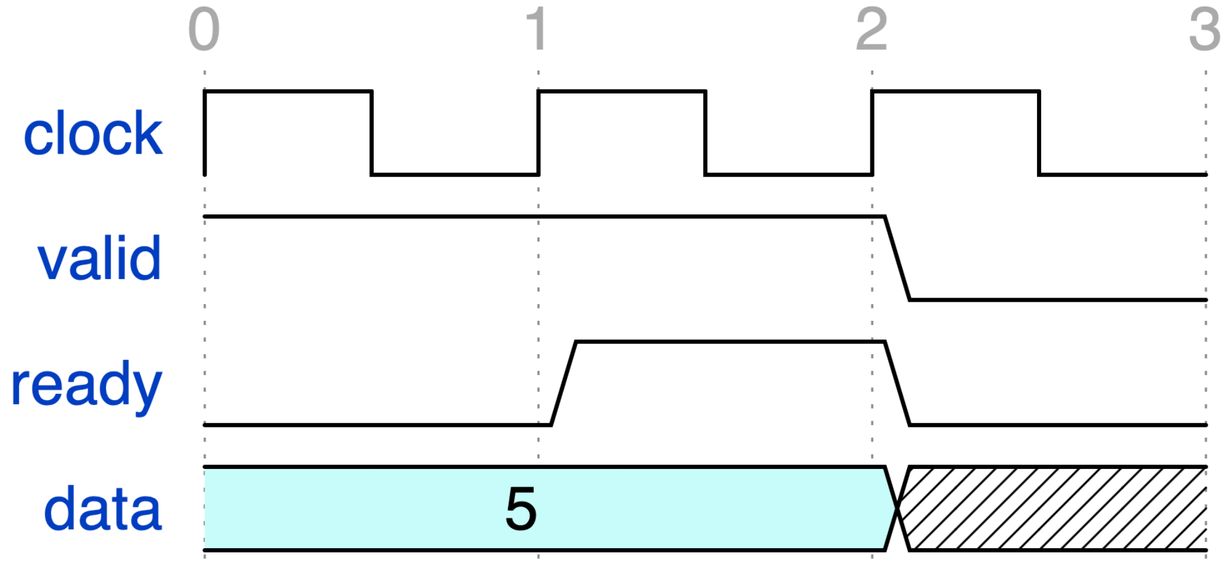


```
prot send_data<DUT: ReadyValid>(
  data: u8
) {
  DUT.valid := 1;
  DUT.data  := data;
  while (DUT.ready ≠ 1) {
    step();
  }
  step();
}
```

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

Finished symbolically executing `send_data`!

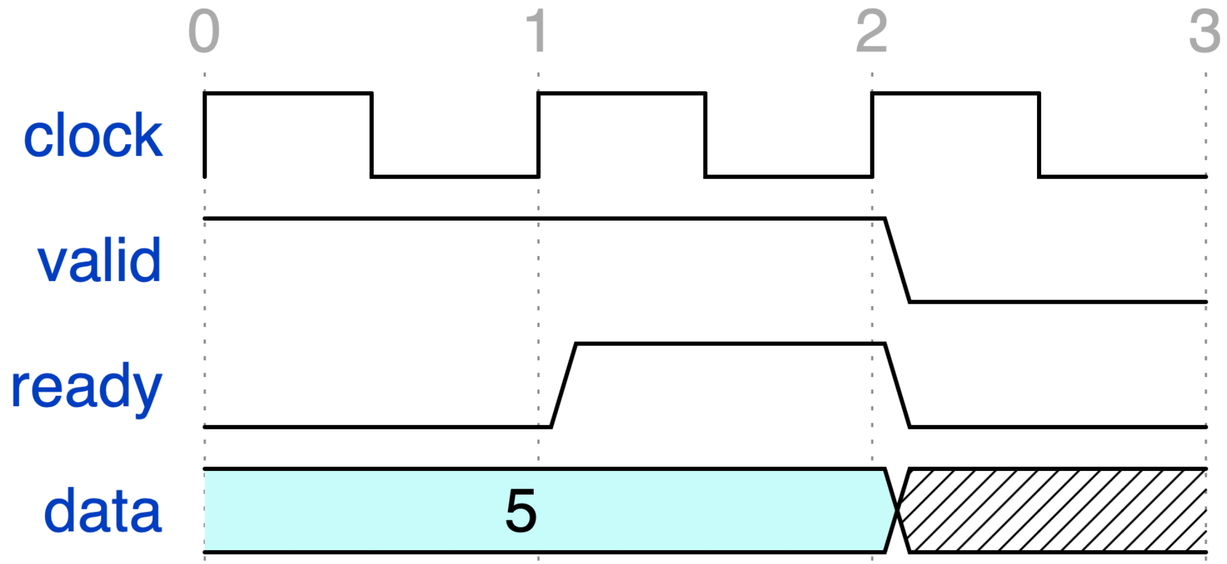
Cycle 2



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

Inferred transaction trace:

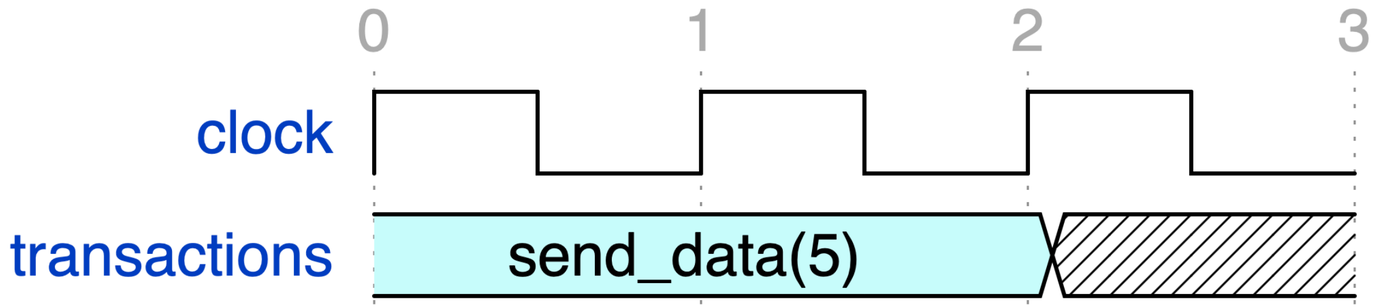
```
send_data(5); // cycles 0-2
```



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

Inferred transaction trace:

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



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)

arm

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)

arm

Wishbone

Open-source interconnect for on-chip communication

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.

ernest
@cs.cornell.edu

cucapra/protocols

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

Ernest
Ng

Nikil
Shyamsunder

Francis
Pham

Adrian
Sampson

Kevin
Laeufer

Thanks!

ernest
@cs.cornell.edu

cucapra/protocols

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

Ernest
Ng

Nikil
Shyamsunder

Francis
Pham

Adrian
Sampson

Kevin
Laeufer