Capra Lab

Specifying Hardware Communication as Programs

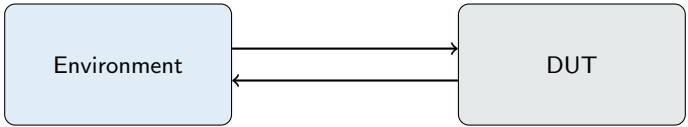
Nikil Shyamsunder

Joint work with Ernest Ng, Francis Pham, Adrian Sampson, Kevin Laeufer

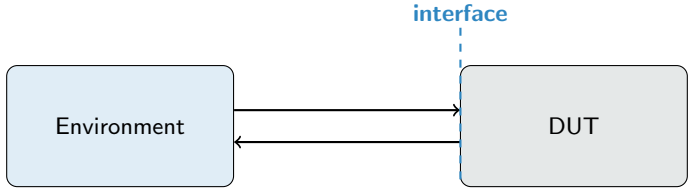
{nvs26, eyn5, fdp25, asampson, laeufer}@cornell.edu

Berkeley, CA · 2026-05-06

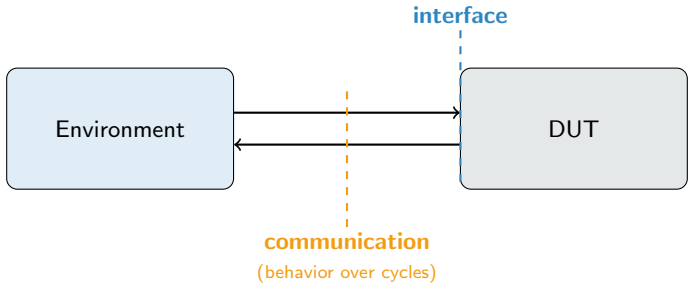
The DUT and its Environment



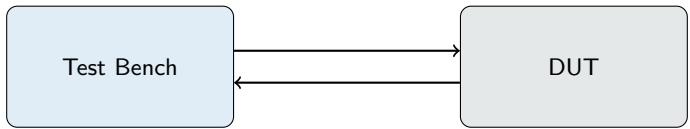
The DUT and its Environment



The DUT and its Environment

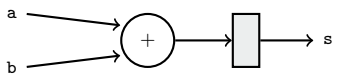


Testing: Replace the Environment



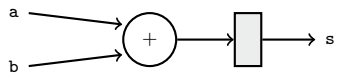
Testing: if we play the role of the environment
and follow the DUT's communication spec,
do we observe expected behavior?

Transactions vs. Signals



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

Transactions vs. Signals



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

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

Specifications are best expressed as **transactions**.

Specifications are best expressed as **transactions**.

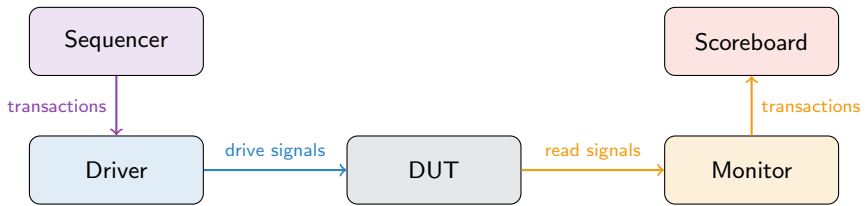
Waveforms live at the level of **cycles and signals**.

Two Kinds of Specification

Functional Specification
what the module computes

Communication Specification
how and when signals move

Testing: UVM Agent Structure



Driver: **transactions** → **trace**. Monitor: **trace** → **transactions**.

The Scattered Communication Specification

- **Driver:** imperative code that drives signals to perform a transaction

The Scattered Communication Specification

- **Driver:** imperative code that drives signals to perform a transaction
- **Monitor:** separate code that reads signals to recover transactions

The Scattered Communication Specification

- **Driver:** imperative code that drives signals to perform a transaction
- **Monitor:** separate code that reads signals to recover transactions
- **Assertion-based verification (SVA):** timing properties as temporal logic

The Scattered Communication Specification

- **Driver:** imperative code that drives signals to perform a transaction
- **Monitor:** separate code that reads signals to recover transactions
- **Assertion-based verification (SVA):** timing properties as temporal logic
- **Neighboring RTL modules** that talk to this block must also respect the spec

The Scattered Communication Specification

- **Driver:** imperative code that drives signals to perform a transaction
- **Monitor:** separate code that reads signals to recover transactions
- **Assertion-based verification (SVA):** timing properties as temporal logic
- **Neighboring RTL modules** that talk to this block must also respect the spec
- **The human debugging a waveform** holds the spec implicitly in their head

The Scattered Communication Specification

- **Driver:** imperative code that drives signals to perform a transaction
- **Monitor:** separate code that reads signals to recover transactions
- **Assertion-based verification (SVA):** timing properties as temporal logic
- **Neighboring RTL modules** that talk to this block must also respect the spec
- **The human debugging a waveform** holds the spec implicitly in their head

No single source of truth $\Rightarrow$ duplicated effort, inconsistencies.

Our Approach

Functional Specification
what the module computes

(out of scope for this work)

Our Approach

Functional Specification
what the module computes

(out of scope for this work)

Communication Specification
how and when signals move

(what we specify)

Our Approach

Functional Specification
what the module computes

(out of scope for this work)

Communication Specification
how and when signals move

(what we specify)

One communication spec ⇒
a single source of truth for many verification applications.

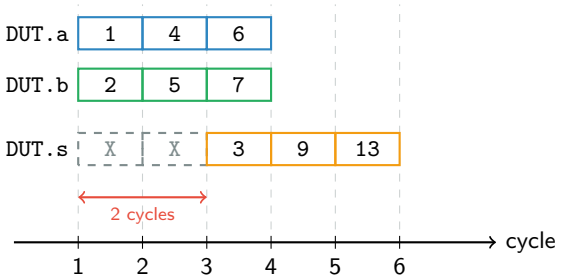
Talk Outline

- The Paso DSL
- Interpreter
- Reconstructor
- Evaluation
- Future Work

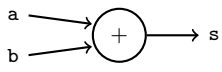
Fixed timing relationship between input and output.

Outputs appear a known number of cycles after inputs.

Running Example: Depth-2 Pipelined Adder

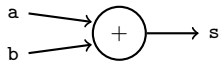


A Protocol Header



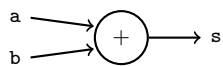
A Protocol Header

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

}
```



A Protocol Header

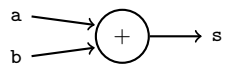
```
prot add_one_two<DUT: Adder>() {

}
```



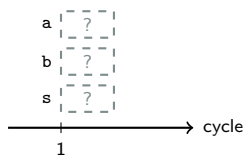
Call: `add_one_two()`

A Protocol Header

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

}
```

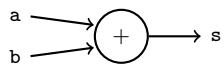


Call: `add_one_two()`

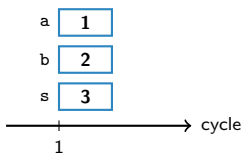


Driving Inputs

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

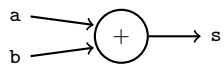


Call: `add_one_two()`

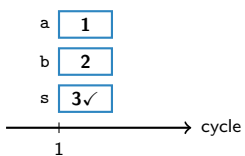


Checking Outputs

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

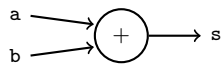


Call: `add_one_two()`

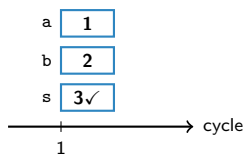


Parameterizing the Protocol

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

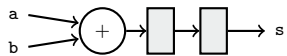


Call: `add(1, 2, 3)`



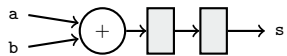
step() — Sequential Circuits

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



step() — Sequential Circuits

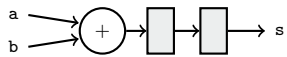
```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  step();
  assert_eq(DUT.s, s);
}
```



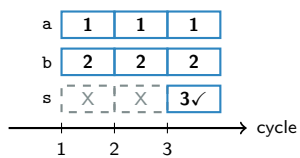
Call: `add(1, 2, 3)`

step() — Sequential Circuits

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

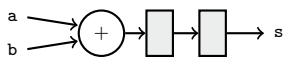


Call: `add(1, 2, 3)`



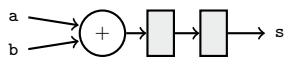
fork() — Pipelining Transactions

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



fork() — Pipelining Transactions

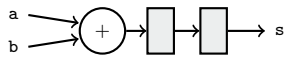
```
prot add<DUT: Adder>(
  a: u32, b: u32, s: u32
) {
  DUT.a := a;
  DUT.b := b;
  step();
  fork();
  step();
  assert_eq(DUT.s, s);
}
```



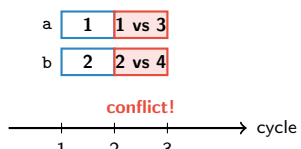
Call: `add(1,2,3); add(3,4,7)`

`fork()` — Pipelining Transactions

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



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



DontCare — Releasing Ports

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

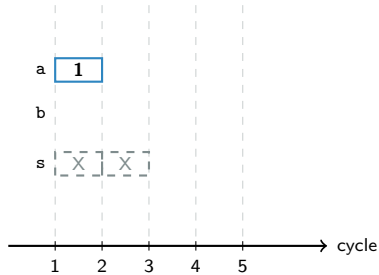
- `X` (DontCare): this port is no longer meaningful for my transaction

Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

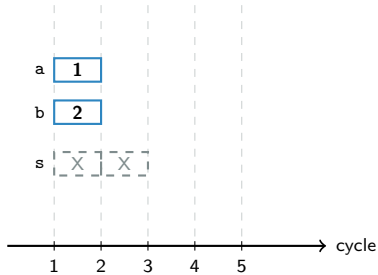


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

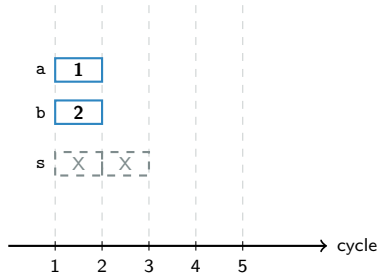


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

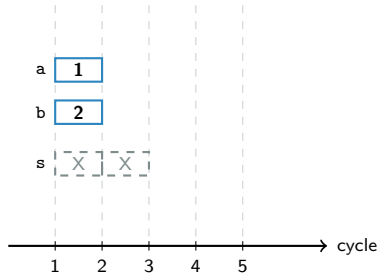


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

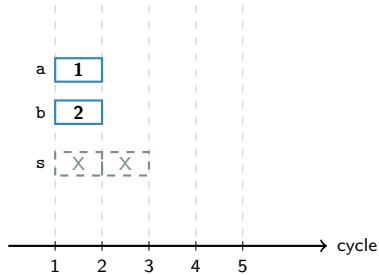


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

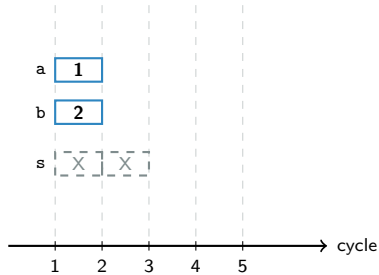


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

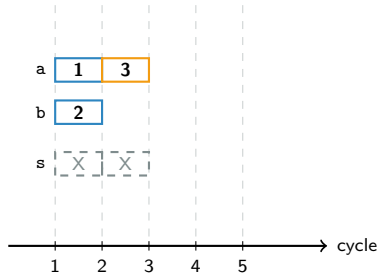


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

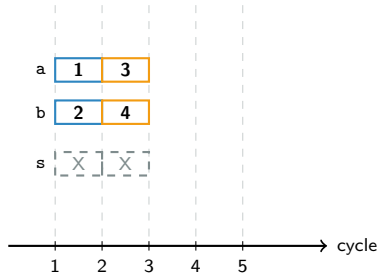


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

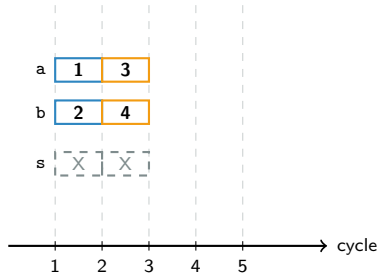


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

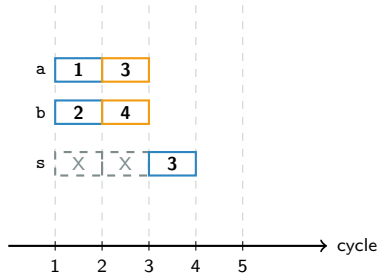


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

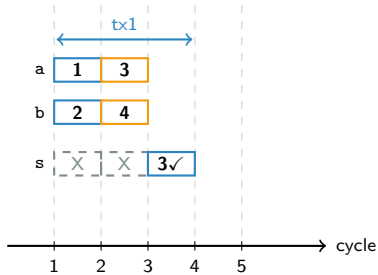


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

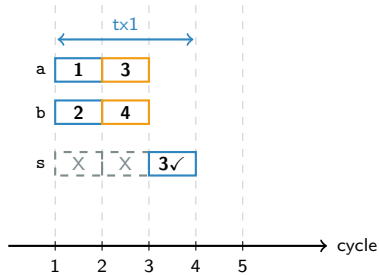


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

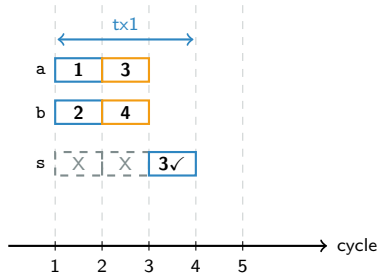


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

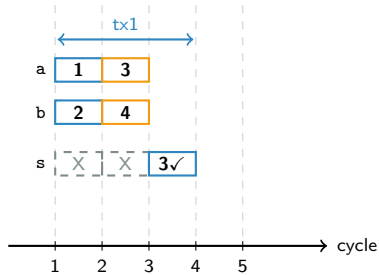


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

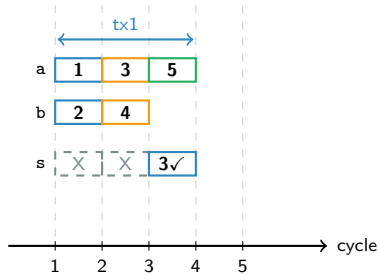


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

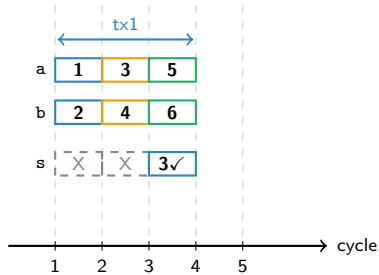


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

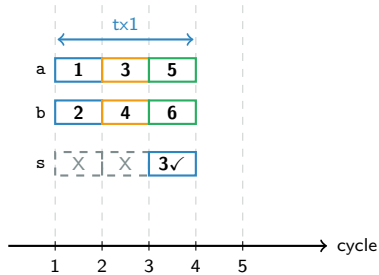


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

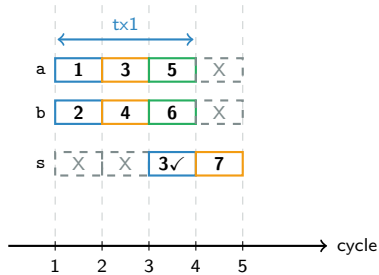


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

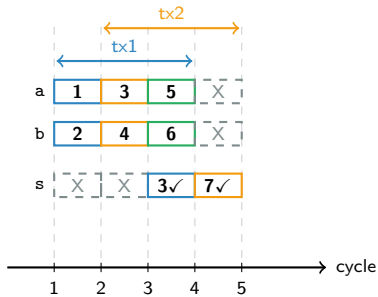


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

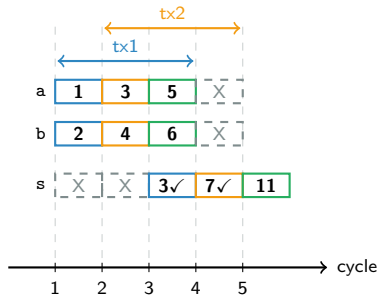


Pipelined Adder: Full Walkthrough

Transaction trace:

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

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

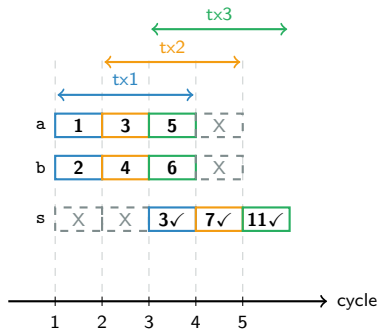


Pipelined Adder: Full Walkthrough

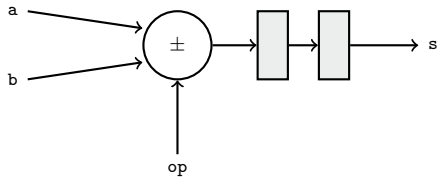
Transaction trace:

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

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



Multiple Protocols on One DUT




```
prot add<DUT: ALU>(
  a: u32, b: u32,
  s: u32
) {
  DUT.a := a; DUT.b := b;
  DUT.op := 1'b0;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork(); step();
  assert_eq(DUT.s, s);
}
```

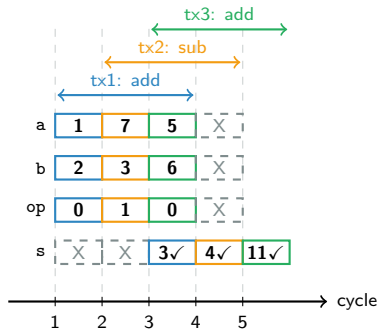
```
prot sub<DUT: ALU>(
  a: u32, b: u32,
  d: u32
) {
  DUT.a := a; DUT.b := b;
  DUT.op := 1'b1;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork(); step();
  assert_eq(DUT.s, d);
}
```

AddSub: Full Walkthrough

Transaction trace:

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

```
prot add<DUT: ALU>(a, b, s) {
  DUT.a := a; DUT.b := b;
  DUT.op := 0;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork(); step();
  assert_eq(DUT.s, s);
}
prot sub<DUT: ALU>(a, b, s) {
  DUT.a := a; DUT.b := b;
  DUT.op := 1;
  step();
  DUT.a := X; DUT.b := X;
  DUT.op := X;
  fork(); step();
  assert_eq(DUT.s, s);
}
```



Explicit synchronization signals
coordinate when data is transferred.

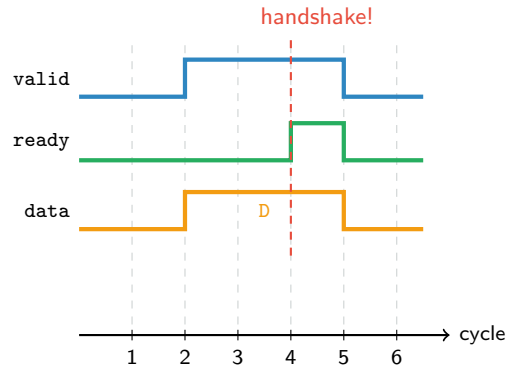
A Latency-Insensitive Interface: Ready-Valid

Interface:

- `valid`: sender signals data is meaningful
- `ready`: receiver signals it can accept
- `data`: payload

Handshake rule:

Data is transferred when `ready` *and* `valid` are both high in the same cycle.



Ready-Valid: Protocol Rules

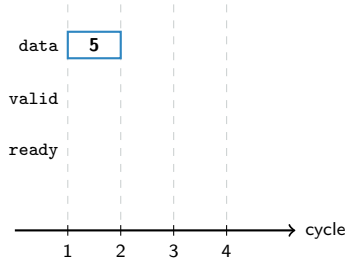
This is the AXI-flavored ready-valid handshake (Chisel's `IrrevocableIO`):

1. **Commitment:** once `valid` is high, it stays high until the handshake completes.
2. **Stability:** `data` stays stable while `valid` is high, waiting for `ready`.

Sending Data: Developing `send_data`

Transaction: `send_data(5)`

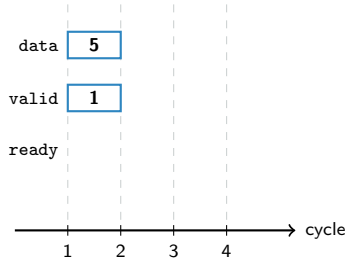
```
prot send_data<DUT: Receiver>(
  data:  u8
) {
  DUT.data := data;
```



Sending Data: Developing `send_data`

Transaction: `send_data(5)`

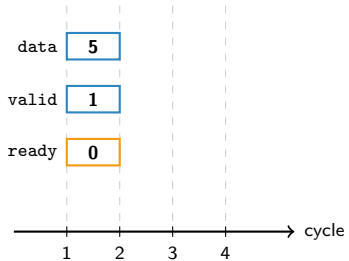
```
prot send_data<DUT: Receiver>(
  data:  u8
) {
  DUT.data := data;
  DUT.valid := 1;
```



Sending Data: Developing `send_data`

Transaction: `send_data(5)`

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

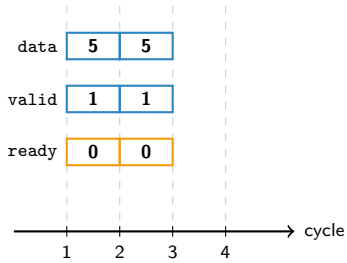


How do we wait until `DUT.ready = 1`?

Sending Data: Developing `send_data`

Transaction: `send_data(5)`

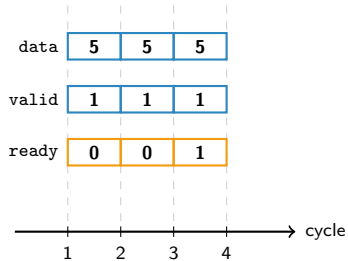
```
prot send_data<DUT: Receiver>(
  data:  u8
) {
  DUT.data := data;
  DUT.valid := 1;
  while (DUT.ready != 1) {
    step();
  }
```



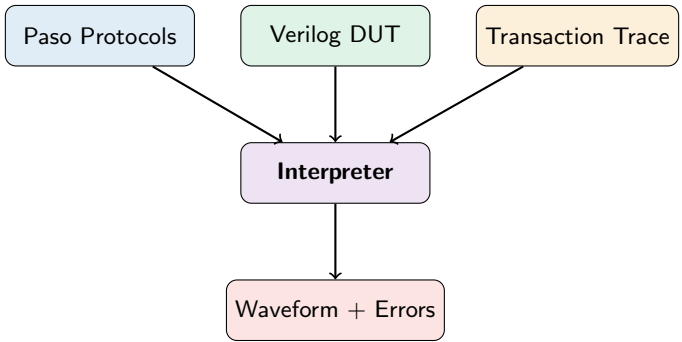
Sending Data: Developing `send_data`

Transaction: `send_data(5)`

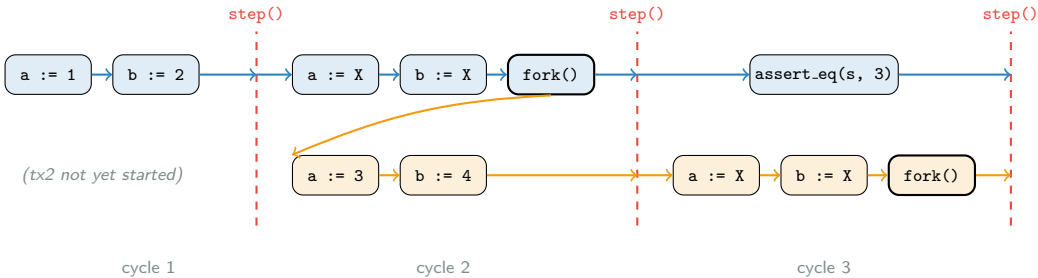
```
prot send_data<DUT: Receiver>(
  data:  u8
) {
  DUT.data := data;
  DUT.valid := 1;
  while (DUT.ready != 1) {
    step();
  }
}
```



Interpreter Overview



Concurrent Execution Model



Why Verilog Testbenches Go Wrong

Hand-written testbenches have two subtle traps:

- **Race conditions:**
  Two concurrent processes assign to the same DUT input in the same cycle. The final value depends on scheduler order.

Why Verilog Testbenches Go Wrong

Hand-written testbenches have two subtle traps:

- **Race conditions:**
  Two concurrent processes assign to the same DUT input in the same cycle. The final value depends on scheduler order.
- **Glitches:**
  A signal changes multiple times within a cycle; a process may sample a transient value that disappears by the clock edge.

Why Verilog Testbenches Go Wrong

Hand-written testbenches have two subtle traps:

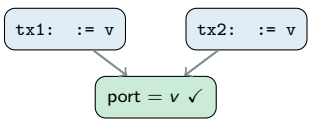
- **Race conditions:**
  Two concurrent processes assign to the same DUT input in the same cycle. The final value depends on scheduler order.
- **Glitches:**
  A signal changes multiple times within a cycle; a process may sample a transient value that disappears by the clock edge.

Paso's execution model rules both out.

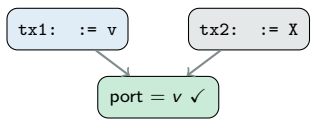
Consistent View of an Input Port

Multiple transactions may touch the same input port each cycle.
They all have to agree on the *final* value.

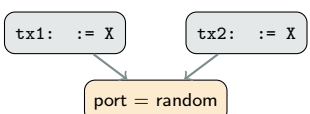
(a) all agree on v



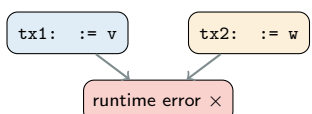
(b) some agree, rest X



(c) all X

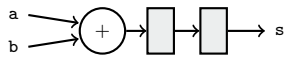


(d) conflict

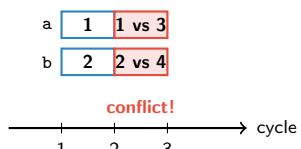


Recall: `fork()` Conflict without DontCare

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



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

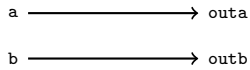


Conflicting Assignment: Real Error

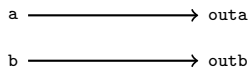
```
prot add<DUT: Adder>(
  a, b, s
) {
  DUT.a := a;
  DUT.b := b;
  step();
  fork();
  // no X assignments
  step();
  assert_eq(s, DUT.s);
  step();
}
```

```
error:  Thread 0 (‘add‘) attempted
  conflicting assignment to ’a’:
  current=4, new=1
   ├ no_dontcare_conflict.prot:12:3
    |
12 |  DUT.a := a;
    |  ^^^^^^^^^^
error:  Thread 1 (‘add‘) attempted
  conflicting assignment to ’a’:
  current=1, new=4
   ├ no_dontcare_conflict.prot:12:3
    |
12 |  DUT.a := a;
    |  ^^^^^^^^^^
Trace 0 execution failed.
```

Two Wires

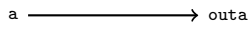


Two Wires




```
prot foo<DUT: TwoWires>() {
  ...
  DUT.b := X;
  step();
  DUT.a := 5;
}
```

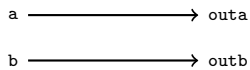
Two Wires




```
prot foo<DUT: TwoWires>() {
  ...
  DUT.b := X;
  step();
  DUT.a := 5;
}
```

```
prot bar<DUT: TwoWires>() {
  ...
  DUT.a := X;
  step();
  DUT.b := DUT.a;
}
```

Two Wires



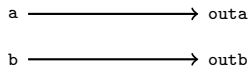

```
prot foo<DUT: TwoWires>() {
  ...
  DUT.b := X;
  step();
  DUT.a := 5;
}
```

```
prot bar<DUT: TwoWires>() {
  ...
  DUT.a := X;
  step();
  DUT.b := DUT.a;
}
```

foo then bar

`DUT.b = 5`

Two Wires




```
prot foo<DUT: TwoWires>() {
  ...
  DUT.b := X;
  step();
  DUT.a := 5;
}
```

```
prot bar<DUT: TwoWires>() {
  ...
  DUT.a := X;
  step();
  DUT.b := DUT.a;
}
```

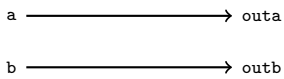
foo then bar

`DUT.b = 5`

bar then foo

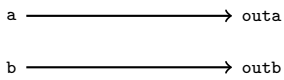
`DUT.b = previous a`

Direct Violation




```
prot foo<DUT: TwoWires>() {
  ...
  DUT.b := X;
  step();
  DUT.a := 5;
}

prot bar<DUT: TwoWires>() {
  ...
  DUT.a := X;
  step();
  DUT.b := DUT.a;
}
```

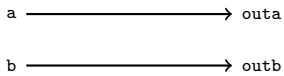
Direct Violation




```
prot foo<DUT: TwoWires>() {
  ...
  DUT.b := X;
  step();
  DUT.a := 5;
}

prot bar<DUT: TwoWires>() {
  ...
  DUT.a := X;
  step();
  DUT.b := DUT.a;
}
```

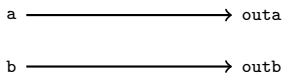
```
error:  Cannot read input 'a'
  which has not been assigned
  a concrete value
   ├ bar.prot:N:M
   |
   |  DUT.b := DUT.a;
   |           ^^^^^
Trace 0 execution failed.
```

Combinational-Dependency Violation




```
prot foo<DUT: TwoWires>() {
  DUT.b := X;
  step();
  DUT.a := 5;
}

prot bar<DUT: TwoWires>() {
  DUT.a := X;
  step();
  DUT.b := DUT.outa;
}
```

Combinational-Dependency Violation




```
prot foo<DUT: TwoWires>() {
  DUT.b := X;
  step();
  DUT.a := 5;
}

prot bar<DUT: TwoWires>() {
  DUT.a := X;
  step();
  DUT.b := DUT.outa;
}
```

```
error:  Cannot observe 'outa'
  whose input 'a' has not been
  assigned a concrete value
   ├─ bar.prot:N:M
   |
   |  DUT.b := DUT.outa;
   |           ^^^^^^^^
Trace 0 execution failed.
```

Rules, Summarized

- Transactions cannot disagree on an input-port value.

Rules, Summarized

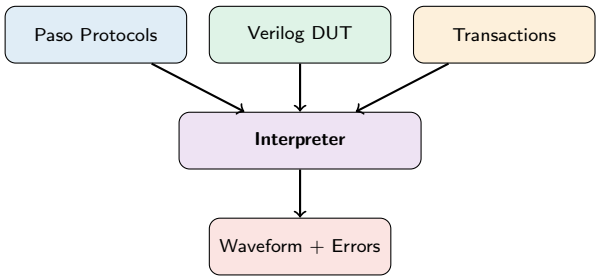
- Transactions cannot disagree on an input-port value.
- A transaction cannot observe an input it has not driven.

Rules, Summarized

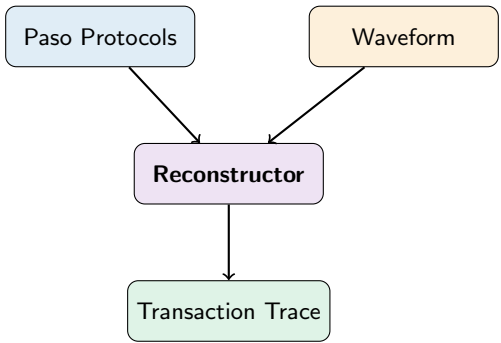
- Transactions cannot disagree on an input-port value.
- A transaction cannot observe an input it has not driven.
- A transaction cannot observe an output whose combinational cone it has not fully driven.

Interpreter: Takeaways

- A protocol can serve as a **driver with extra checks**.
- Race conditions and glitches are ruled out by **runtime checks**.



Reconstructor: Motivation



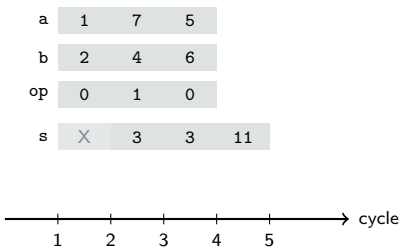
Goal: Given protocols + a waveform, infer the *transaction trace* consistent with the waveform.

Reconstructor: Use Cases

- Overlay transaction-level information in your waveform viewer.
- Print the transaction trace directly in your terminal.
- Feed the trace to a Python script for post-processing or triage.
- Jump from a waveform cursor to the matching protocol call during debugging.

Reconstructor: AddSub Example

Given: protocols `add`, `sub` (`op=0` / `op=1`) and a waveform.



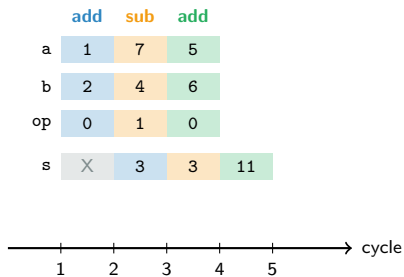
Reconstructor: AddSub Example

Given: protocols `add`, `sub` (`op=0` / `op=1`) and a waveform.

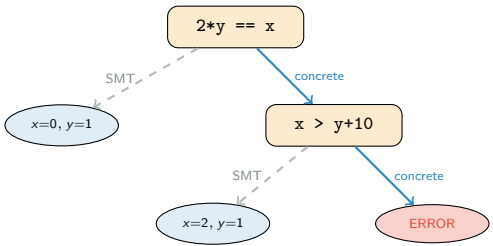
Inferred trace:

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

Reconstructor determines *which* protocol and *what* arguments produced each waveform segment.



Dynamic Symbolic Execution



DSE finds assertion failures by trying every branch it can reach.

Reverse-Executing a Function with DSE

You can also use DSE to **solve for the inputs of a function**. Negate the desired outcome as an assertion:

```
let r = foo(x);
assert!(r != 5, "found an input x such that foo(x) == 5");
```

Paso Reconstruction as DSE

Reconstruction is DSE applied to the protocol, using the waveform as the concrete path.

Symbolic part:

- Which transaction starts at each `fork()` point

Paso Reconstruction as DSE

Reconstruction is DSE applied to the protocol, using the waveform as the concrete path.

Symbolic part:

- Which transaction starts at each `fork()` point
- The argument values of that transaction

Paso Reconstruction as DSE

Reconstruction is DSE applied to the protocol, using the waveform as the concrete path.

Symbolic part:

- Which transaction starts at each `fork()` point
- The argument values of that transaction

Concrete part (known from the waveform):

- Every DUT input and output at every cycle

Paso Reconstruction as DSE

Reconstruction is DSE applied to the protocol, using the waveform as the concrete path.

Symbolic part:

- Which transaction starts at each `fork()` point
- The argument values of that transaction

Concrete part (known from the waveform):

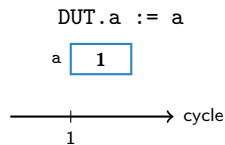
- Every DUT input and output at every cycle

Unlike classical DSE, we enumerate *all* consistent traces.

Well-Formedness Makes Solving Trivial

Every constraint we generate has **at most one unknown**:

- A single bit of a parameter equals 0 or 1.



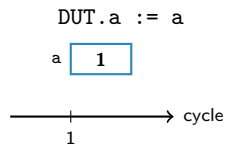
Constraint: $a = 1$

Well-Formedness Makes Solving Trivial

Every constraint we generate has **at most one unknown**:

- A single bit of a parameter equals 0 or 1.

No SMT solver needed.



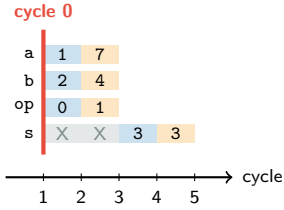
Constraint: $a = 1$

Walkthrough: Pipelined ALU

Two protocols on a 2-cycle-latency ALU:

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

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

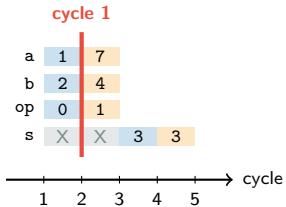


Infer the transaction trace that produced this waveform.

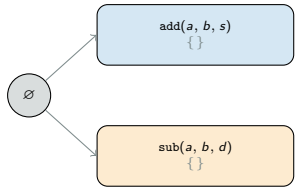
ALU **Cycle 1**: Spawn Two Execution Paths

Protocols (PC ▶)

```
prot add(a,b,s):          prot sub(a,b,d):
> DUT.a:=a                > DUT.a:=a
  DUT.b:=b                  DUT.b:=b
  DUT.op:=0                 DUT.op:=1
  step()                    step()
  ...                       ...
```



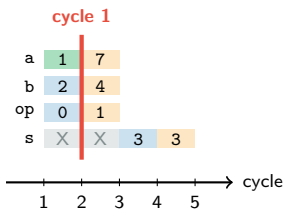
Exploration tree



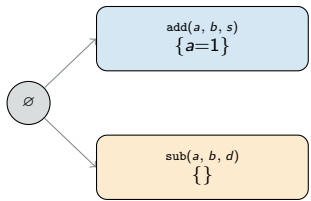
One execution path per protocol; constraints empty.

Interpreting add: `DUT.a := a`

```
prot add(a,b,s):          prot sub(a,b,d):
> DUT.a:=a                 DUT.a:=a
  DUT.b:=b                 DUT.b:=b
  DUT.op:=0                DUT.op:=1
  step()                   step()
```

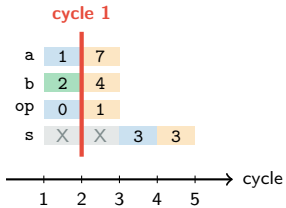


Exploration tree

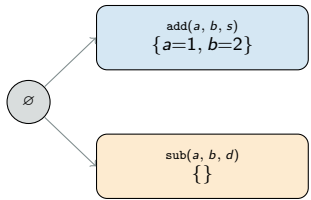


Interpreting add: `DUT.b := b`

```
prot add(a,b,s):          prot sub(a,b,d):
  DUT.a:=a                  DUT.a:=a
> DUT.b:=b                  DUT.b:=b
  DUT.op:=0                 DUT.op:=1
  step()                    step()
```

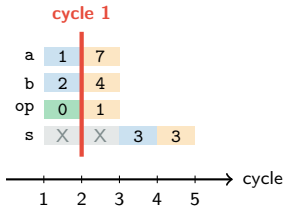


Exploration tree

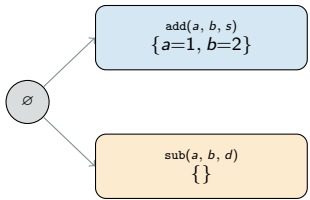


Interpreting add: DUT.op := 0

```
prot add(a,b,s):          prot sub(a,b,d):
  DUT.a:=a                  DUT.a:=a
  DUT.b:=b                  DUT.b:=b
> DUT.op:=0                 DUT.op:=1
  step()                    step()
```



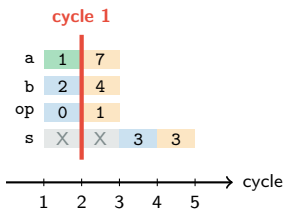
Exploration tree



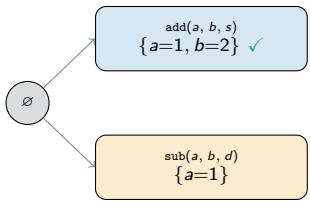
Constraint: $0 = 0$ ✓

Interpreting sub: DUT.a := a

```
prot add(a,b,s):          prot sub(a,b,d):
  DUT.a:=a                > DUT.a:=a
  DUT.b:=b                  DUT.b:=b
  DUT.op:=0                 DUT.op:=1
  step()                    step()
```

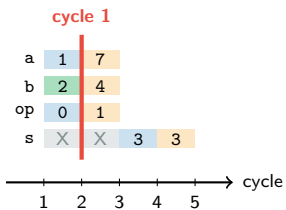


Exploration tree

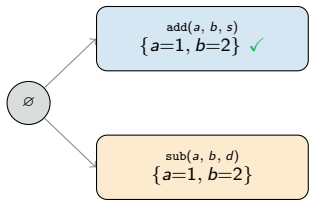


Interpreting sub: DUT.b := b

```
prot add(a,b,s):          prot sub(a,b,d):
  DUT.a:=a                  DUT.a:=a
  DUT.b:=b                > DUT.b:=b
  DUT.op:=0                 DUT.op:=1
  step()                    step()
```

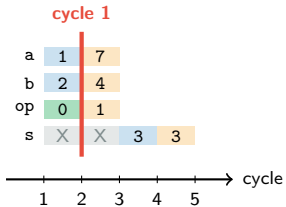


Exploration tree

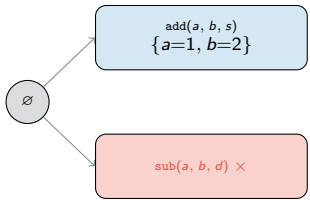


Interpreting sub: DUT.op := 1

```
prot add(a,b,s):          prot sub(a,b,d):
  DUT.a:=a                  DUT.a:=a
  DUT.b:=b                  DUT.b:=b
  DUT.op:=0               > DUT.op:=1
  step()                    step()
```



Exploration tree



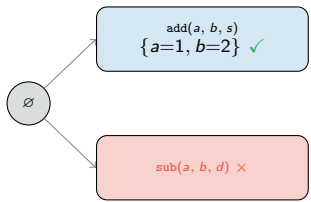
Constraint: $1 = 0$? ×

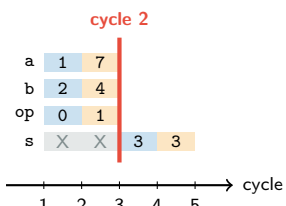
Interpreting add: step()

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

```
prot sub(a,b,d):
  DUT.a:=a
  DUT.b:=b
  DUT.op:=1
  step()
```

Exploration tree





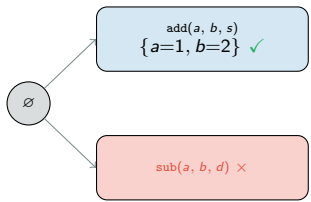
Advance to cycle 2. sub pruned.

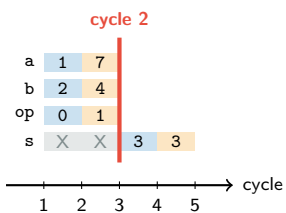
Interpreting add: DUT.a := X

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

```
prot sub(a,b,d):
  DUT.a:=a
  DUT.b:=b
  DUT.op:=1
  step()
```

Exploration tree





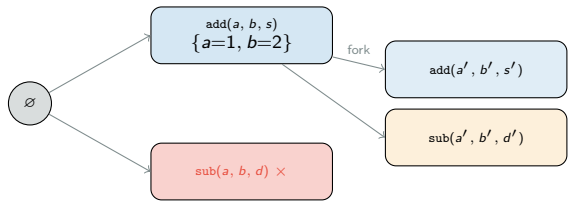
DontCare releases inputs. No constraint on cy2 a.

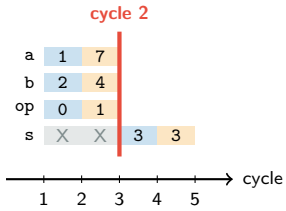
Interpreting add: `fork()`

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

```
prot sub(a,b,d):
 DUT.a:=a
 DUT.b:=b
 DUT.op:=1
 step()
```

Exploration tree

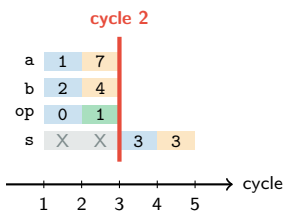




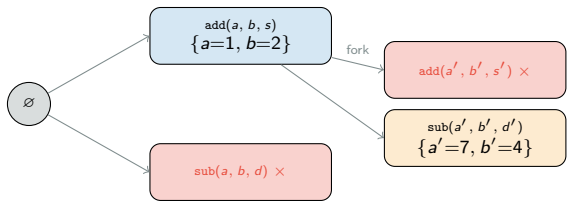
`fork()` spawns one new execution path per protocol.

Interpreting the New Transaction (skip ahead)

```
prot add(a,b,s):          prot sub(a,b,d):
  DUT.a:=a                  DUT.a:=a
  DUT.b:=b                  DUT.b:=b
  DUT.op:=0               > DUT.op:=1
  step()                    step()
```



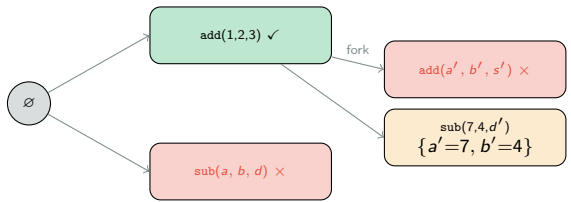
Exploration tree

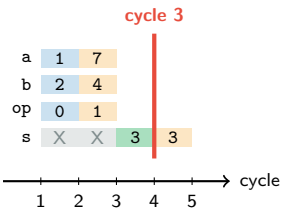


Interpreting add: assert_eq(DUT.s, s)

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

Exploration tree



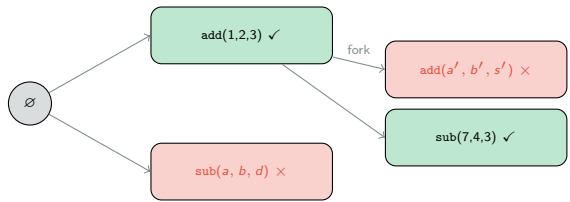


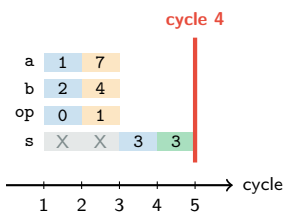
add completes: binds s=3.

Interpreting sub: assert_eq(DUT.s, d)

```
prot sub(a,b,d):
  DUT.a:=a
  DUT.b:=b
  DUT.op:=1
  step()
  DUT.a:=X; DUT.b:=X
  DUT.op:=X
  fork()
  step()
> assert_eq(DUT.s,d)
```

Exploration tree (final)





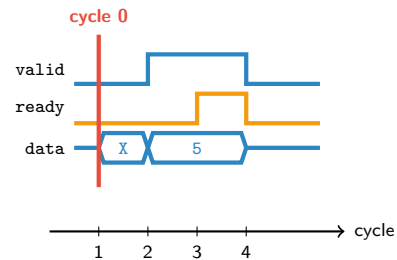
Inferred trace: `add(1,2,3); sub(7,4,3);`

Walkthrough: `send_data`

Two protocols on a ready-valid receiver:

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

```
prot idle<DUT: Receiver>() {
  DUT.valid := 0;
}
```

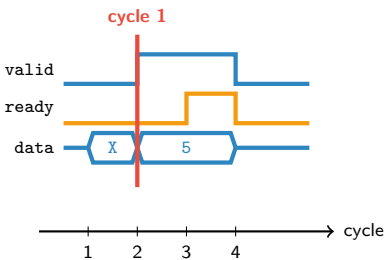


send_data Cycle 1: idle

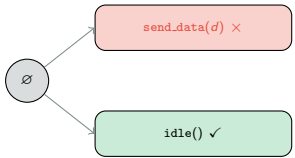
Protocols

```
prot send_data(data):
> DUT.data := data
  DUT.valid := 1
  while (ready != 1):
    step()
```

```
prot idle():
  DUT.valid := 0
```



Exploration tree

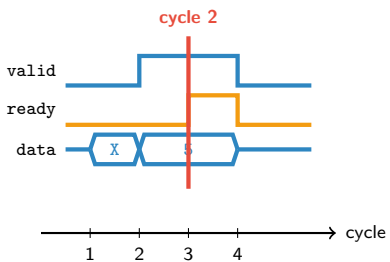


send_data fails (valid=0); idle matches.

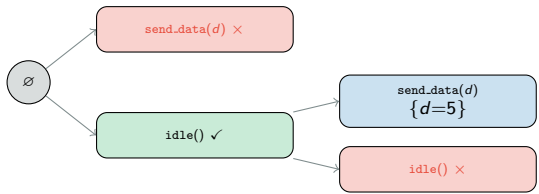
`send_data` **Cycle 2**: Spawn + Infer *d*

Protocol (PC ▶)

```
prot send_data(data):
  DUT.data := data
  DUT.valid := 1
> while (ready != 1):
    step()
```



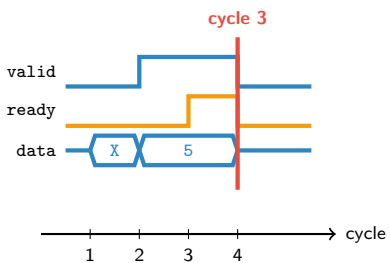
Exploration tree



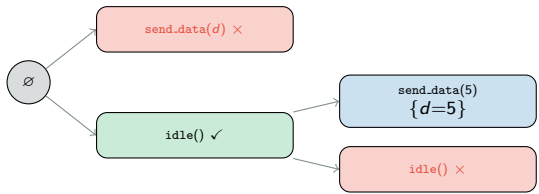
send_data **Cycle 3**: Loop Persists

Protocol (PC ▶)

```
prot send_data(data=5):
  DUT.data := data
  DUT.valid := 1
> while (ready != 1):
    step()
```



Exploration tree

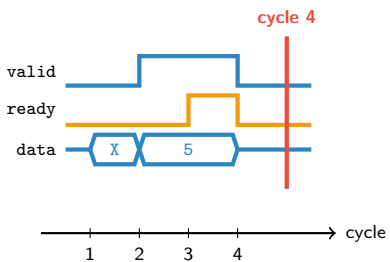


Trace still `ready=0` ⇒ loop again.

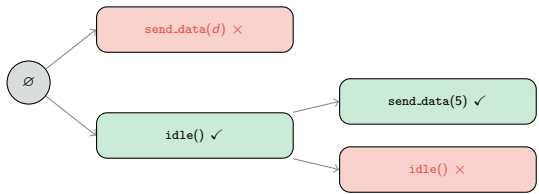
`send_data` **Cycle 4**: Handshake & Complete

Protocol (PC ▶)

```
prot send_data(data=5):
  DUT.data := data
  DUT.valid := 1
  while (ready != 1):
    step()
> }       (handshake!)
```



Exploration tree (final)



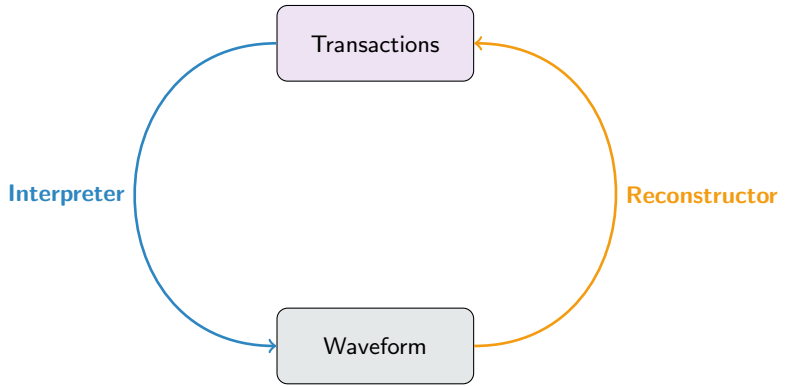

```
// trace 0
trace {
idle();            // [time:  0ns -> 10ns]
send_data(5);  // [time:  10ns -> 40ns]
}
```

When the Monitor Fails

If the waveform doesn't match any protocol, the monitor reports a violation — and points at the statement it couldn't satisfy.

```
// trace 0
trace {
idle();           // [time:  0ns -> 10ns]
send_data(5);  // [time:  10ns -> 25ns]
}

error:  [send_data] executing step 1 of the transaction:  0x1 != 0x0
├─ send_data.prot:5:3
|
5 |  DUT.valid := 1;
|  ^^^^^^^^^^^^^^ [send_data] executing step 1 of
|               the transaction:  0x1 != 0x0
```

Round-Trip: Interpreter ↔ Reconstructor



Writing **one Paso spec** allows us to go **both** ways!

Evaluation

(ongoing)

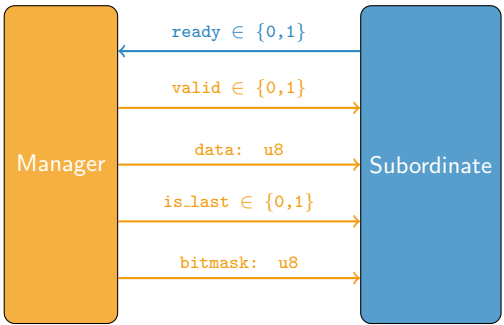
Evaluation: Two Goals

1. **Expressivity.**
   One Paso protocol drives and monitors real hardware designs without modification.

2. **Bug-catching.**
   A protocol catches communication-level bugs in waveforms.

AXI-Stream

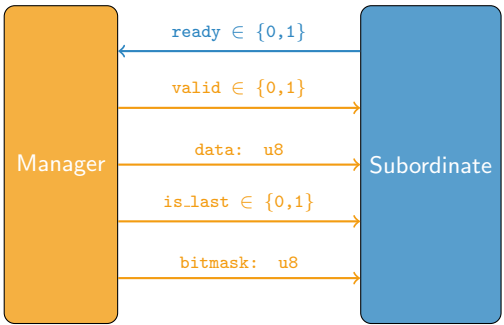
ARM ready-valid interface for stream processing



Two On-Chip Interconnects

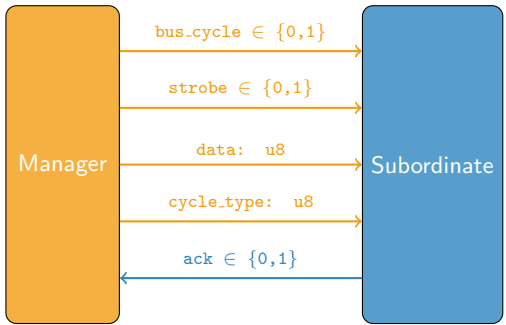
AXI-Stream

ARM ready-valid interface for stream processing



Wishbone

Open-source interconnect for on-chip communication



Expressivity Case Study: Wishbone

We're *in the process* of validating that one Paso spec can cover real hardware:

- Open-source memory-bus protocol (FOSSi Foundation)

Expressivity Case Study: Wishbone

We're *in the process* of validating that one Paso spec can cover real hardware:

- Open-source memory-bus protocol (FOSSi Foundation)
- Collecting Wishbone FPGA designs from GitHub

Expressivity Case Study: Wishbone

We're *in the process* of validating that one Paso spec can cover real hardware:

- Open-source memory-bus protocol (FOSSi Foundation)
- Collecting Wishbone FPGA designs from GitHub
- Target: **one** Paso spec that drives *and* monitors all of them

Protocol Bugs in the Wild

FPGA bug benchmark (`efeslab/hardware-bugbase`):

- Each bug ships with paired *buggy* + *fixed* Verilog

Protocol Bugs in the Wild

FPGA bug benchmark (`efeslab/hardware-bugbase`):

- Each bug ships with paired *buggy* + *fixed* Verilog
- Reproduce bugs, collect both waveforms

Protocol Bugs in the Wild

FPGA bug benchmark (`efeslab/hardware-bugbase`):

- Each bug ships with paired *buggy* + *fixed* Verilog
- Reproduce bugs, collect both waveforms
- Hand-write a Paso protocol for each interface

Protocol Bugs in the Wild

FPGA bug benchmark (`efeslab/hardware-bugbase`):

- Each bug ships with paired *buggy* + *fixed* Verilog
- Reproduce bugs, collect both waveforms
- Hand-write a Paso protocol for each interface

Reconstructor on *fixed* waveform:	✓ consistent trace inferred
Reconstructor on *buggy* waveform:	× error: unable to infer transactions

AXI-Stream Bugs Caught

We're *in the process* of checking 4 AXI-Stream bugs
the reconstructor can already report.

- Unstable signals during stalls in data transfer

AXI-Stream Bugs Caught

We're *in the process* of checking 4 AXI-Stream bugs
the reconstructor can already report.

- Unstable signals during stalls in data transfer
- Buffer overflow

AXI-Stream Bugs Caught

We're *in the process* of checking 4 AXI-Stream bugs
the reconstructor can already report.

- Unstable signals during stalls in data transfer
- Buffer overflow
- Missing ACKs for requests

AXI-Stream Bugs Caught

We're *in the process* of checking 4 AXI-Stream bugs
the reconstructor can already report.

- Unstable signals during stalls in data transfer
- Buffer overflow
- Missing ACKs for requests
- Missing signals during packet transmission

In each case, the waveform is inconsistent with the protocol specification.

Future Work

Expand the DSL

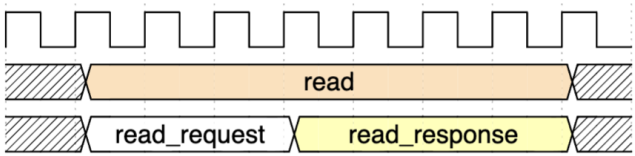
- **AXI / AXI-Lite**
  Composition of protocols and reasoning across multiple channels.

Expand the DSL

- **AXI / AXI-Lite**
  Composition of protocols and reasoning across multiple channels.

- **Wishbone burst operations**
  Multi-beat transfers need an *unbounded sequence* type and iteration over it.

Transaction-Level Waveform Viewing



Advancing the Compiler

A compiler that turns protocol specs into synthesizable RTL unlocks several downstream capabilities:

- **On-FPGA runtime monitoring.**
  Catch communication errors in place, or stream transactions off-chip for post-hoc debugging.

Advancing the Compiler

A compiler that turns protocol specs into synthesizable RTL unlocks several downstream capabilities:

- **On-FPGA runtime monitoring.**
  Catch communication errors in place, or stream transactions off-chip for post-hoc debugging.
- **Bounded model checking.**
  Prove that a DUT cannot violate its protocol within k cycles.

Advancing the Compiler

A compiler that turns protocol specs into synthesizable RTL unlocks several downstream capabilities:

- **On-FPGA runtime monitoring.**
  Catch communication errors in place, or stream transactions off-chip for post-hoc debugging.
- **Bounded model checking.**
  Prove that a DUT cannot violate its protocol within k cycles.
- **SVA equivalence.**
  Automatically find waveform traces where a Paso protocol and an SVA spec disagree.

Advancing the Compiler

A compiler that turns protocol specs into synthesizable RTL unlocks several downstream capabilities:

- **On-FPGA runtime monitoring.**
  Catch communication errors in place, or stream transactions off-chip for post-hoc debugging.
- **Bounded model checking.**
  Prove that a DUT cannot violate its protocol within k cycles.
- **SVA equivalence.**
  Automatically find waveform traces where a Paso protocol and an SVA spec disagree.
- **Test-bench code generation.**
  One protocol, many ecosystems: cocotb, UVM, chiseltest.

cucapra/protocols nvs26@cornell.edu

Write **one protocol specification**
that informs **all communication-related details**
of the hardware design process.


Nikil Shyamsunder


Ernest Ng


Francis Pham


Adrian Sampson


Kevin Laeufer

cucapra/protocols nvs26@cornell.edu

Write **one protocol specification**
that informs **all communication-related details**
of the hardware design process.

Thank you!


Nikil Shyamsunder


Ernest Ng


Francis Pham


Adrian Sampson


Kevin Laeufer

References

- J. Ma, G. Zuo, K. Loughlin, H. Zhang, A. Quinn, B. Kasikci. *Debugging in the Brave New World of Reconfigurable Hardware.* ASPLOS 2022. [methodology]
- `efeslab/hardware-bugbase` — Kasikci Lab, U. Michigan. [bug collection]
- `alexforencich/verilog-axis` — open-source Verilog AXI-Stream components. [original bug sources]