Building an Interpreter for an Imperative Hardware Interface Specification Language

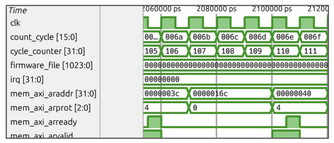
Nikil Shyamsunder, Francis Pham, Ernest Ng, Adrian Sampson, Kevin Laeufer
nvs26@cornell.edu

Cornell Bowers CIS College of Computing and Information Science

Repository

1. Hardware Debugging is Hard

What transactions are running when?



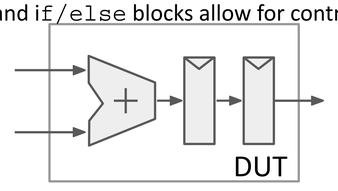
- Most hardware is interacted with via some set of protocols, which defines its interface.
- This interface specification is implicitly scattered across ad-hoc assertions and test harnesses. **There is no single source of truth.**
- Result: Bugs (deadlocks, signal asynchrony, etc.)

2. Our Solution: A Hardware Interface Specification Language

For many testing tasks, it is useful to separate the communication protocol from the contents of what is being communicated.

Protocols allows us to do this:

- `symbol := RHS` assigns the value of the `RHS` expression to the DUT input port `symbol`. The right-hand side expression may be arbitrarily chosen, represented by `X`.
- `step(n)` advances the clock by `n` cycles.
- `fork()` allows for concurrent protocol execution.
- `assert_eq(e1,e2)` tests equality between `e1` & `e2`.
- `while` and `if/else` blocks allow for control flow.




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

Protocol

3. An Interpreter for Protocols

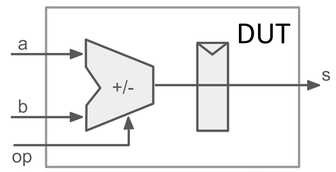
```
fn add<DUT: AddSub>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b; DUT.op := 0;
  step();
  DUT.a := X; DUT.b := X; DUT.op := X;
  fork();
  assert_eq(DUT.s, s);
}
```

Protocols

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

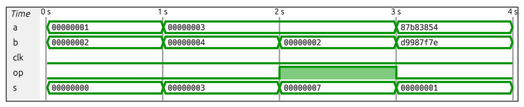
Transactions

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

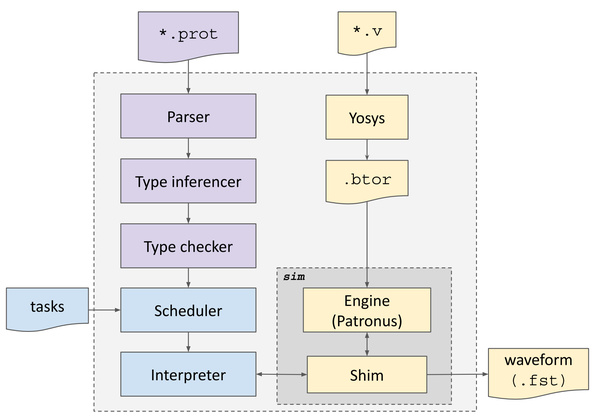


Interpreter

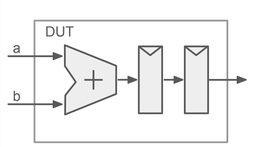
Waveform



4. Implementing an Interpreter



5. Concurrency in Protocols with `fork()`



time	1	2	3	4	5
DUT.a	a'	a''	a'''		
DUT.b	b'	b''	b'''		
DUT.s	X	X	s'	s''	s'''

```
fn add<DUT: Adder>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b; DUT.op := 0;
  step();
  DUT.a := X; DUT.b := X; DUT.op := X;
  fork();
  step();
  DUT.a := X; DUT.b := X; DUT.op := X;
  assert_eq(DUT.s, s);
}
```

`fork()` allows the next transaction (if one exists) to be started this cycle.

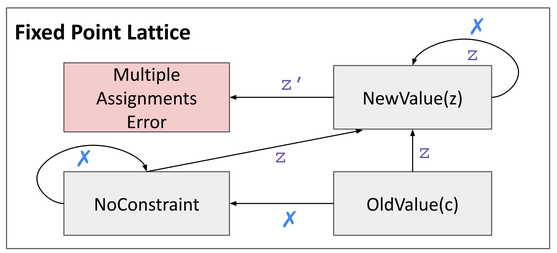
6. Assignment Validity

Two types of assignments:

1. **Concrete:** `dut.a := a`
2. **Don't Care:** `dut.a := X`

Three states of DUT Input Values:

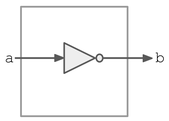
1. **OldValue:** value retained from prior cycle of circuit
2. **NoConstraint:** value does not matter, i.e. setting the value randomly would not change the outcome of *all* transactions
3. **NewValue:** value assigned by some thread before executing next `step();`



Fixed-Point Iteration: Run all active transactions between their previous `step()` and their next `step()` call, ignoring all assertions. Once all assignments become stable (no input values change), convergence is achieved. Run transactions a final time, checking assertions against the now stable DUT's input port state.

7. Fixed-Point on Combinational Inverter

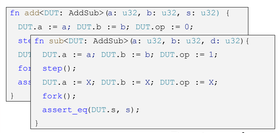
```
fn invert<DUT: inverter>() {
  DUT.a := 0;
  step();
  DUT.a := DUT.b;
}
```



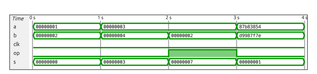
(Cycle 2) Iteration	1	2	3
(in) DUT.a	OldValue(0)	NewValue(1)	Error
(out) DUT.b	1	0	
Assignment	DUT.a := 1	DUT.a := 0	

Fixed-Point can catch errors in Protocols, even in single-threaded cases.

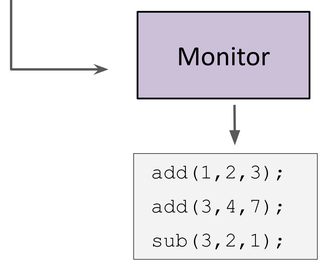
8. Application: Monitor



Protocols



Waveform




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

Transactions Executed

The Monitor can determine whether the waveform reflects the developers intended specification, without info about the RTL implementation. We can perform round-trip testing between the Monitor and Interpreter.

9. Application: Constraint Loosening

```
fn identity1
<DUT: Identity>(a, s) {
  DUT.a := a;
  step();

  fork();
  DUT.a := a;
  step();

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

```
fn identity2
<DUT: Identity>(a, s) {
  DUT.a := a;
  step();

  fork();
  DUT.a := X;
  step();

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

`identity2` admits a superset of the traces admitted by `identity1`. Can we find the most general protocol which is correct for the DUT?