Scoreboarding - Transaction-based Verification

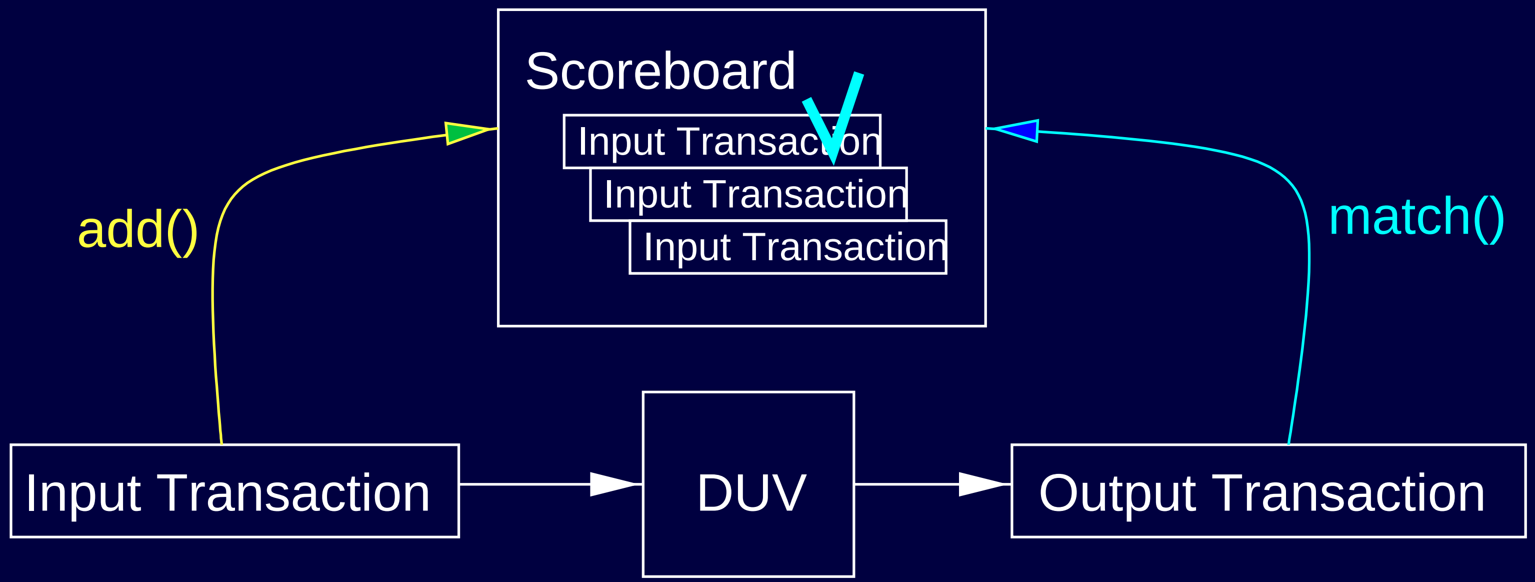
Put created transactions/packets on the **scoreboard.**

Compare emerging transactions/packets.

- Collect output transactions/packets.
- Find corresponding item on scoreboard & delete. Use unique IDs.

Account for transactions/packets that disappear.

At the end, check that scoreboard is empty.



Scoreboarding (simple) example in e - I

Assume: DUV does not change order of packets.

- Hence, first packet on scoreboard has to match received packet.

```
import packet_s;
unit scoreboard {
  !expected_packets : list of packet_s;

  add_packet(p_in : packet_s) is {
    expected_packets.add(p_in);
  };

  check_packet(p_out : packet_s) is {
    var diff : list of string;
    -- Compare physical fields of first packet on scb with p_out.
    -- Report up to 10 differences.
    diff = deep_compare_physical(expected_packets[0], p_out, 10);
    check that (diff.is_empty())
      else dut_error(‘‘Packet not found on scoreboard’’, diff);
    -- If match was successful, continue.
    out(‘‘Found received packet on scoreboard.’’);
    expected_packets.delete(0);
  };
};
```

Scoreboarding (simple) example in e - II

Recording a packet on the scoreboard

Extend driver such that:

- When packet is driven into DUV call **`add_packet`** method of scoreboard.

  ⇒ Current packet is copied to scoreboard.

  ☼ Useful to define an **event** that indicates when packet is being driven!

Checking for a packet on the scoreboard:

Extend receiver such that:

- When a packet was received from DUV call **`check_packet`**.

  ⇒ Try to find the matching packet on scoreboard.

  ☼ Useful to define an **event** that indicates when a packet is being received!

Advanced Constraints

keep *constraint-bool-expr*; where *constraint-bool-expr* is a simple or compound Boolean expression.

- States restriction on the values generated for fields in the struct.

```
keep kind!=tx or len==16;
```

- Describes required relationships between field values and other struct items.

```
struct packet {
    kind : [tx, rx];
    len : int;
    keep kind == tx => len==16;
  --when tx packet { keep len == 16; }; exactly same effect
};
```

Hard constraints are applied when the enclosing struct is generated. If constraints can't be met, generator issues **constraint contradiction message.**

Biased pseudo-random Generation

Using **`keep soft`** (e.g. to set default values) and **`select`**:

- ```
  struct transaction {
    address : uint;
    keep soft address == select {
      10: [0..49];
      60: 50;
      30: [51..99];
    };
  };
  ```

NOTE: *Soft constraints* can be overridden by hard constraints!

- ```
  extend instruction {
    keep soft op_code == select {
      40: [ADD, ADDI, SUB, SUBI];
      20: [XOR, XORI];
      10: [JMP, CALL, RET, NOP];
    };
  };
  ```

In practice, getting the weights/bias right (for coverage closure) requires significant engineering skill.

Generation with keep

Generation order is important: It influences the distribution of values!

- ```
  struct packet {
          kind : [tx, rx];
          length : byte;
          keep length>15 => kind==rx;
  };
  ```

1. If **`kind`** is generated first, **`kind`** is **`tx`** about half the time because there are only two legal values for kind.
2. If **`length`** is generated first, the distribution is different.

- Consider using: **`keep gen (kind) before (length);`**
  ```

Randomized Test Generation needs...

...repeatability:

Same testbench version + same test

+ same random seed

= same stimulus data.

Is this all? The testbench evolves over time!

and random stability:

Changes to the testbench should not affect **orthogonal** aspects!

- Packet data structure:

```
struct packet {
...
payload: list of byte;
...};
```

Randomized Test Generation needs...

...repeatability:

Same testbench version + same test

+ same random seed

= same stimulus data.

- **Is this all? The testbench evolves over time!**

and random stability:

- Changes to the testbench should not affect **orthogonal** aspects!
- Packet data structure with interrupted field:

```
struct packet {
...
payload: list of byte;
interrupted: bool;
...};
```

- With same seed should give the same **`payload`** data!

Advanced Techniques: SN temporal checking

SN Temporal Language

- Capture behaviour over time for synchronization with DUV, functional coverage and **protocol checking.**

- Language consists of:
  - temporal expressions (TEs)
  - temporal operators
  - **`event`** struct members to define occurrences of events during sim run
  - **`expect`** struct members for checking temporal behaviour

NEW: PSL/Sugar compatible expressions (more later).

Temporal Expressions in e

- Each TE is associated with a **sampling event.**
- Sampling event indicates when the TE should be evaluated by SN.

Syntax examples:

- true(boolean-exp)@sample-event
- rise/fall/change(expression)@sample-event

Events in SN

- Events are used to **synchronize with the DUV** or to debug a test.
- **Events are struct members.**

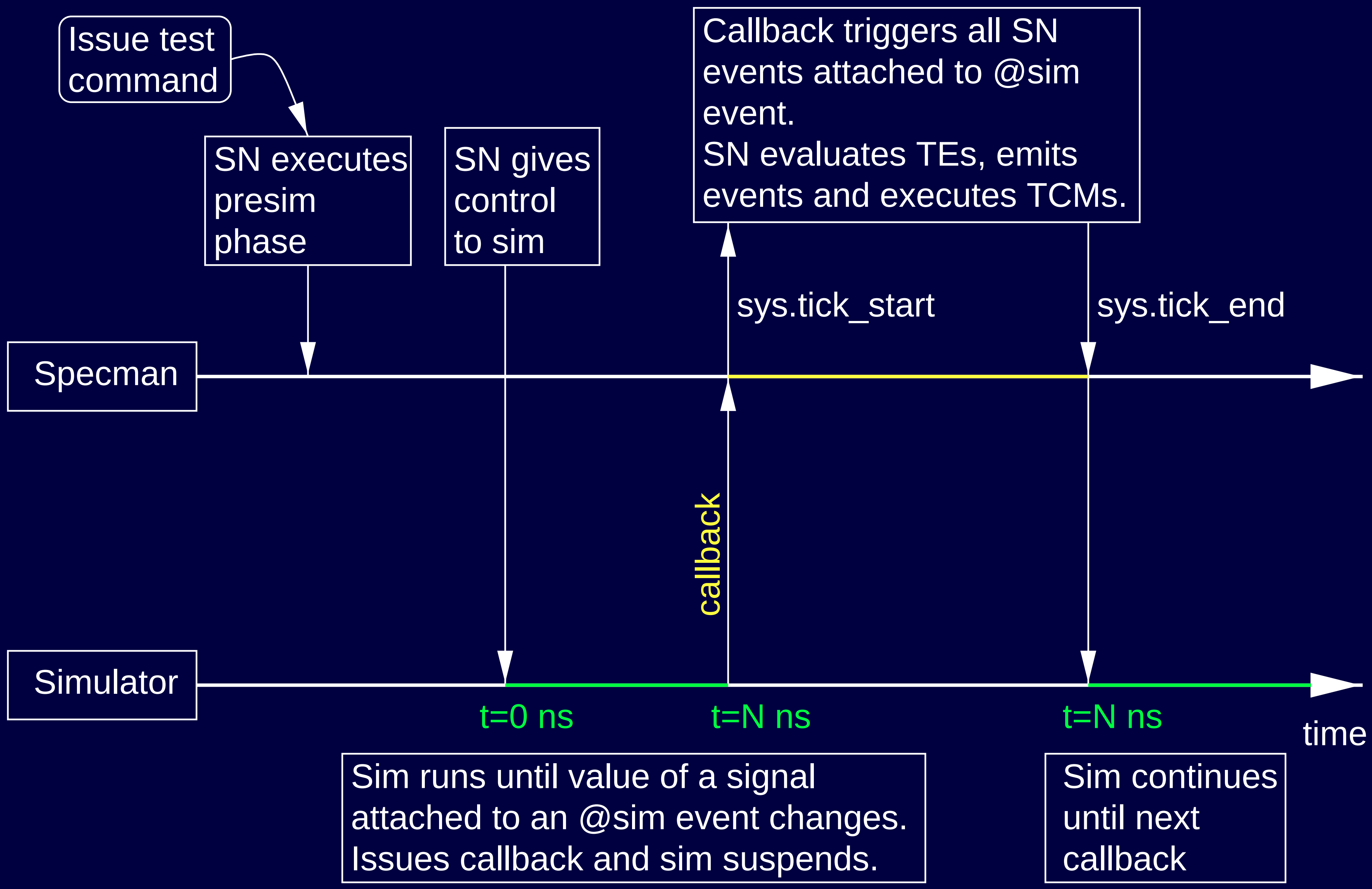
- **Automatic emission** of events:

```
<'
  extend driver_s {
     event clk is fall('calc1_sn.c_clk') @sim;
     event resp is change('calc1_sn.out_resp1')@clk;
  };
'>
```

- **Explicit emission** of event:

```
<'
  extend driver_s {
     collect_response(cmd : command_s) @clk is also {
        emit cmd.cmd_complete;
     };
  };
'>
```

Synch between SN and Simulator



SN Predefined Event: @sim

- `event clk is rise ('~/top/clk') @sim;`

`@sim` is **special sampling event** occurring at any simulator callback.

- Expression must be an HDL signal path in the simulated model.

Signal does not have to be a clock.

- No restriction for signal to be periodic or synchronous.

Might slow down simulation!

Clock signal can also be emitted from e code and driven into DUV. (But usually more efficient to generate clock in HDL.)

When not running with a simulator attached to SN, use **`@sys.any`**.

Conforming to Stimulus Protocol

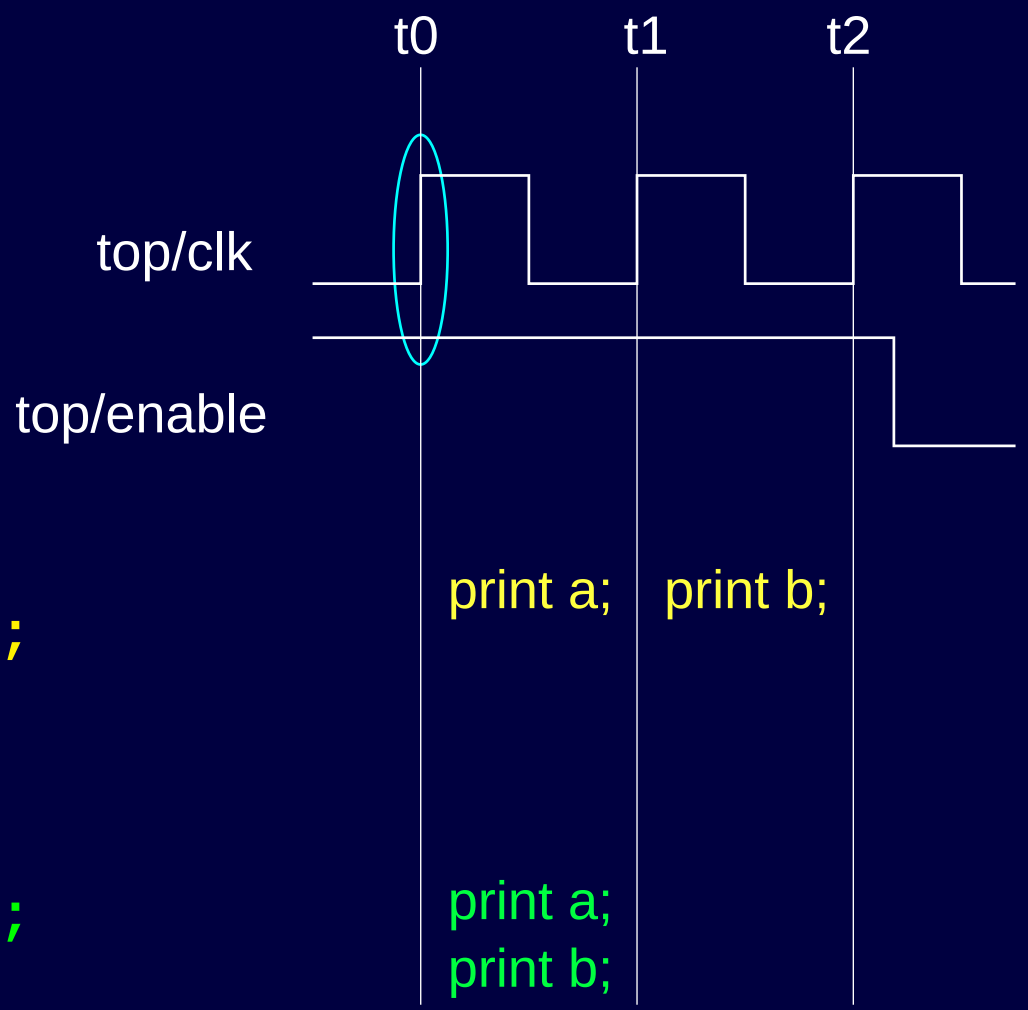
Must be able to react to state of DUV during simulation!

- clock, signal changes, sequences of events

e language provides **`wait`** (till next cycle) and **`sync`** actions which allow to pause procedural code until event occurs.

```
print a;
wait true('~top/enable'==1)@clk;
print b;
```

```
print a;
sync true('~top/enable'==1)@clk;
print b;
```



Methods with a Notion of Time

TCMs - Time Consuming Methods

- Depend on sampling event.
- Can be executed over several simulation cycles.

```
collect_response(cmd : command_s) @clk is {
   wait @resp; -- wait for the response
   cmd.resp = 'calc1_sn.out_resp1';
   cmd.dout = 'calc1_sn.out_data1';
}; // collect_response
```

- Implicit **sync** action at beginning of TCM.
- TCM must be called or started to execute.

```
run() is also {
   start drive();        // spawn
}; // run
```

- Non-TCMs can't *call* TCMs because they have no notion of time.
- TCMs can (only) be *started* (using `start`) from a non-TCM!

Temporal Checking Methodology

1. Capture important DUV temporal behaviour with events and TEs.
2. Use **`expect`** struct members to declare temporal checks.

Syntax: **`expect`** TE **`else dut_error(`***string***`);`**

Example temporal checks:

```
expect @req => {[..4];@ack} @clk
else dut_error("Acknowledge did not follow
                request within 1 to 5 clock cycles.");

expect @buffer_full =>  eventually  @int @clk
else dut_error("Buffer full, but interrupt did not occur.");
```

- **`eventually`** Sometime *before the end of simulation!*