DOULOS KnowHow TUTORIAL

Delivering KnowHow www.doulos.com

Practical Asynchronous SystemVerilog Assertions

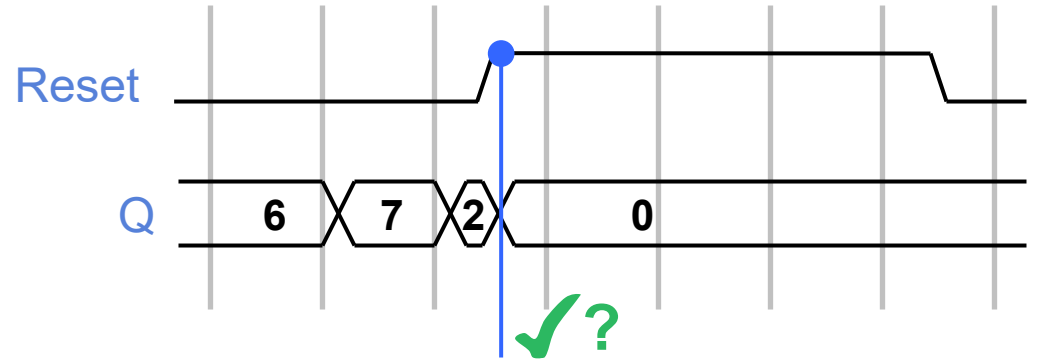
Presenter: Doug Smith
Engineer / Instructor

Asynchronous Assertions

```
module Counter ( input Clock, Reset, Enable,
Load, UpDn, input [7:0] Data, output [7:0] Q );

always @( posedge Reset or posedge Clock )
   if ( Reset )
      Q <= 0;
   else
      if ( Enable )
         if ( Load )
            Q <= Data;
         else
            if ( UpDn )
              Q <= Q + 1;
            else
              Q <= Q - 1;
endmodule
```

```
initial begin
   ... Reset = 1;
```

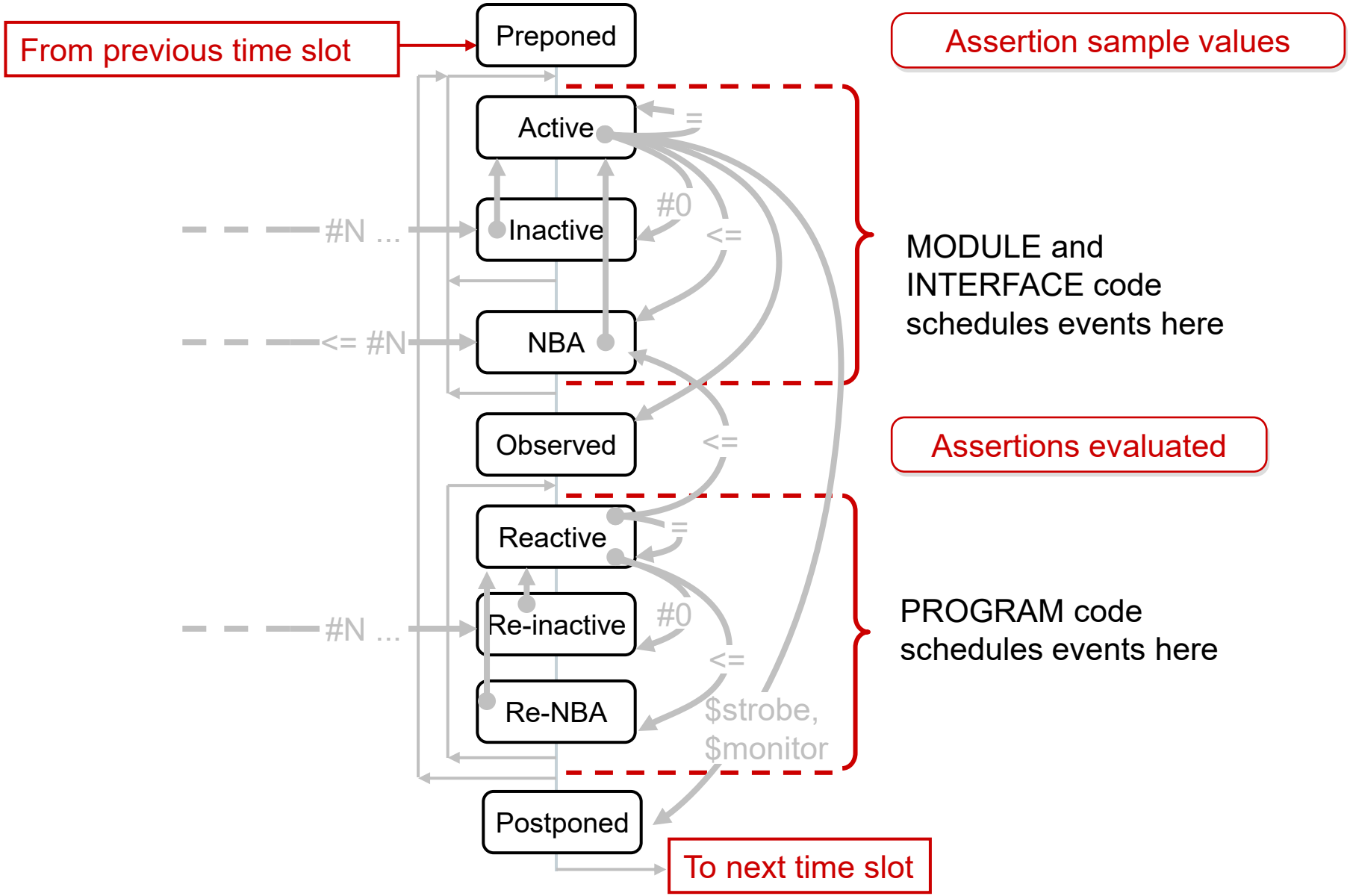
```
initial forever
   Clock = #5 ~Clock;
```



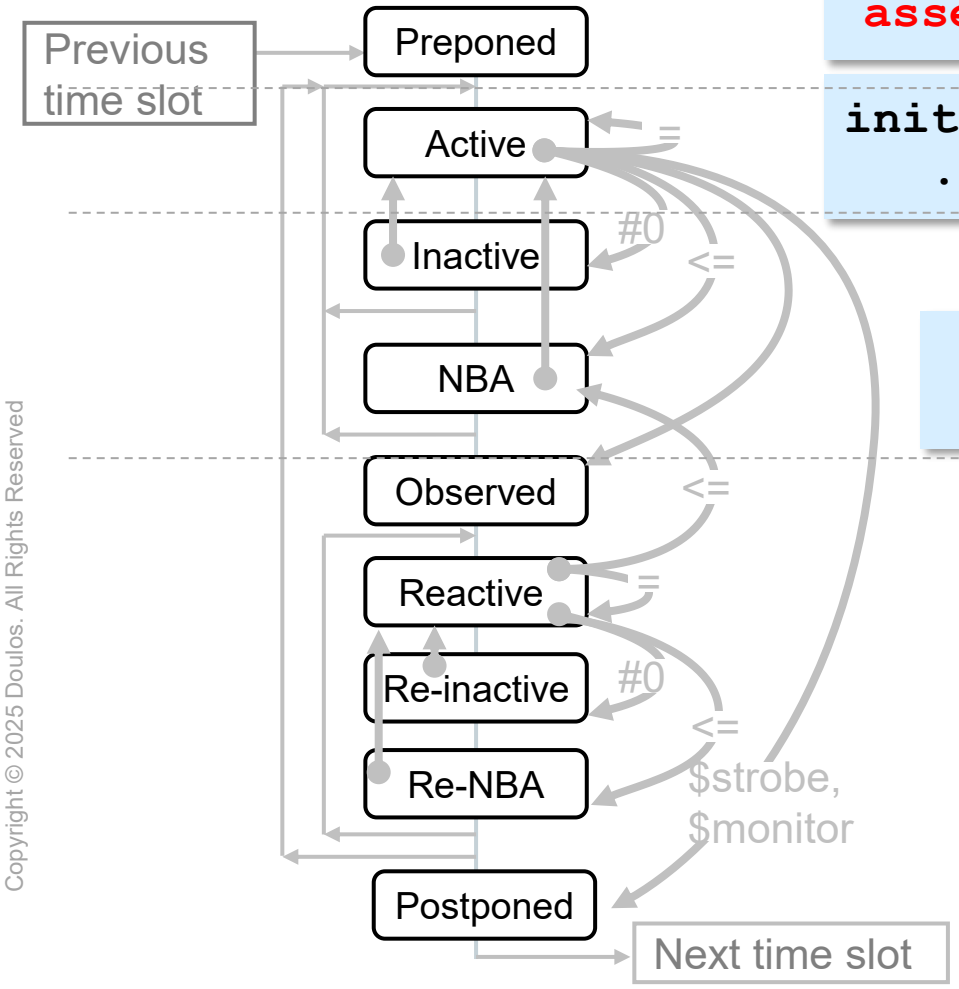

```
assert property ( @(posedge Reset) Q == 0 );
```

Will this work?

SystemVerilog Scheduler



Difficulty with Async Checking




```
assert property ( @(posedge Reset) Q == 0 );
```

```
initial begin
   ... Reset = 1;
```

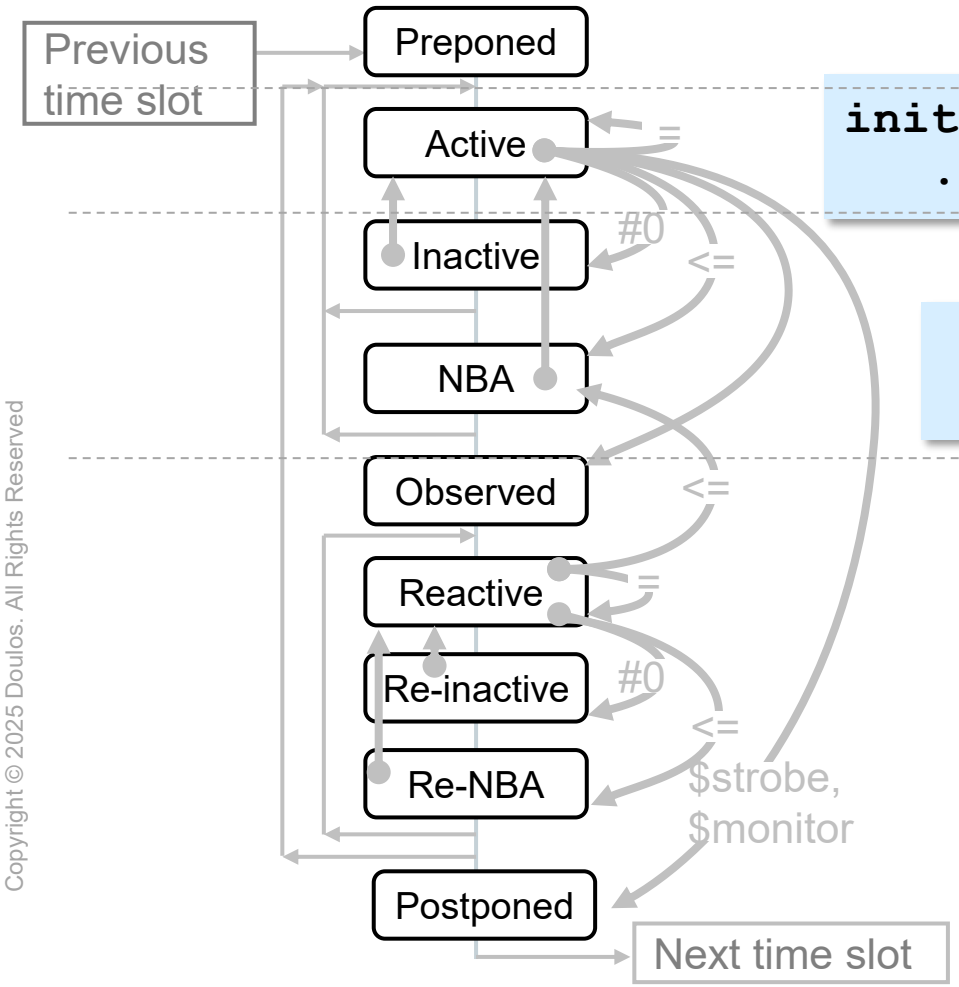
```
initial forever
   Clock = #5 ~Clock;
```

```
always @( posedge Reset ...)
   if ( Reset ) Q <= 0; ...
```

Reset	Q
0	Qprev
1	Qprev
1	0
1	0

RTL not updated yet!

Difficulty with Async Checking




```
initial begin
    ... Reset = 1;
```

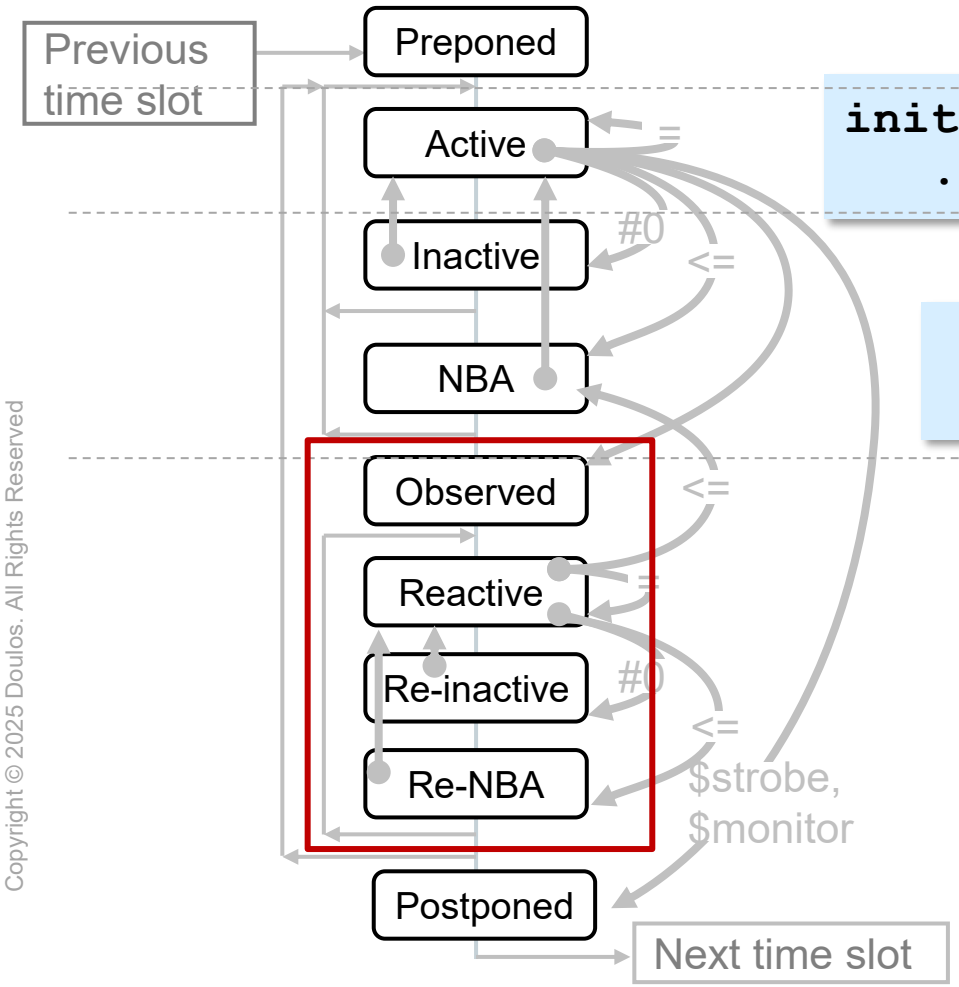
```
always @(posedge Reset)
    assert ( Q == 0 );
```

```
always @( posedge Reset ...)
    if ( Reset ) Q <= 0; ...
```

Reset	Q
0	Qprev
1	Qprev
1	0
1	0

RTL not updated yet!

Difficulty with Async Checking




```
initial begin
    ... Reset = 1;
```

```
always @( posedge Reset ...)
    if ( Reset ) Q <= 0; ...
```

Reset	Q
0	Qprev
1	Qprev
1	0
1	0

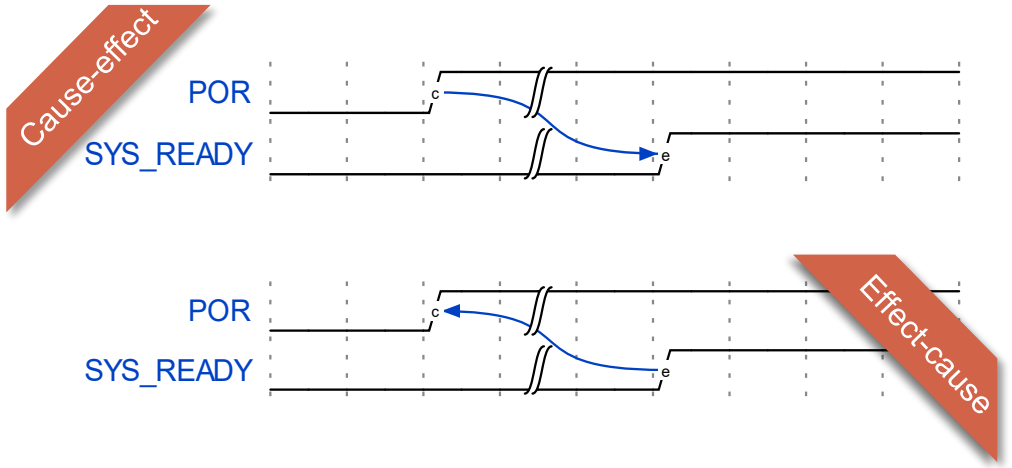
Checking needs delayed

Requirements for Async Assertions

Portable across simulators

Deterministic

Thorough – checks in both directions



In other words, ***practical asynchronous assertions***

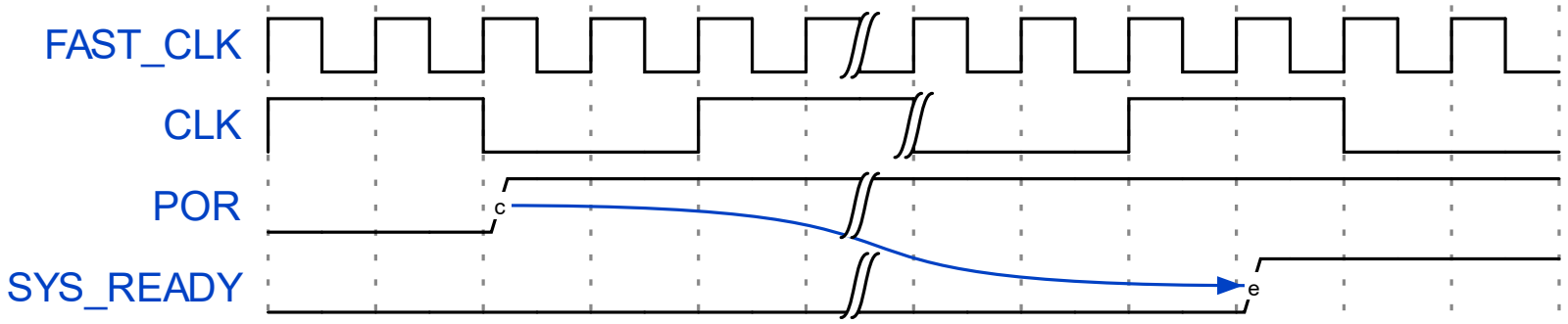
See DVCon 2010 paper for other solutions:

"Asynchronous Behaviors Meet Their Match with SystemVerilog Assertions"

https://www.doulos.com/knowhow/systemverilog/asynchronous-behaviors-meet-their-match-with-systemverilog-assertions/

Common Methods for Asynchronous Checking

Synchronous, oversampling, or fast clock




```
default clocking cb @(posedge CLOCK); endclocking
```

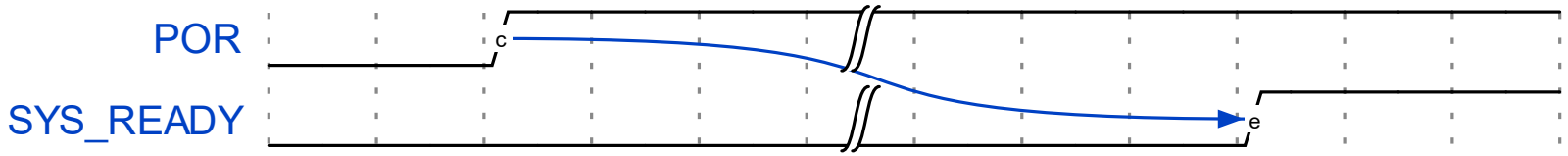
OR

```
default clocking cb @(posedge FAST_CLK); endclocking
```

```
assert property ( $rose(POR) |-> ##[0:$] SYS_READY );
```

What about glitches?

Event based methods




```
bit cover_por = 0;
cover property ( @(posedge POR) 1 ) cover_por = 1;
```

Then …

```
assert property ( @(posedge SYS_READY) cover_por );
```

What if SYS_READY never occurs?

Pros and Cons

Oversampling

- Pro – works with any verification flow (sim, emulation, formal, prototyping)
- Con – glitchy RTL behavior may go undetected

Coverage

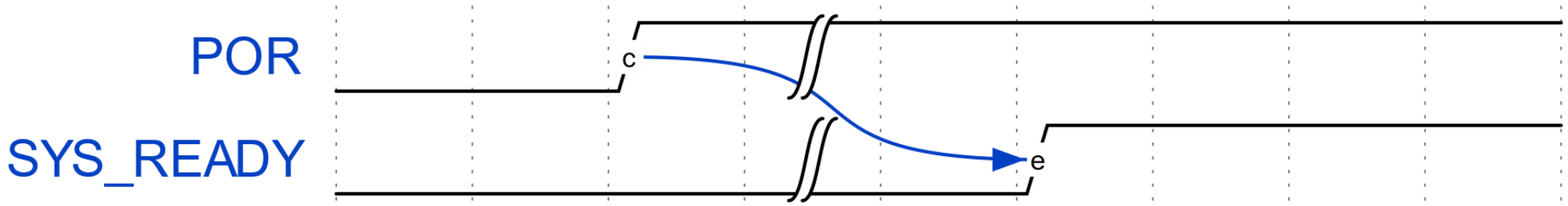
- Pro – easy to write sophisticated scenarios
- Con – overlapping events undetected or event never occurs

Solution? Delay assertion checking …

Cause and Effect

Scenario 1

Async signal causes another async event

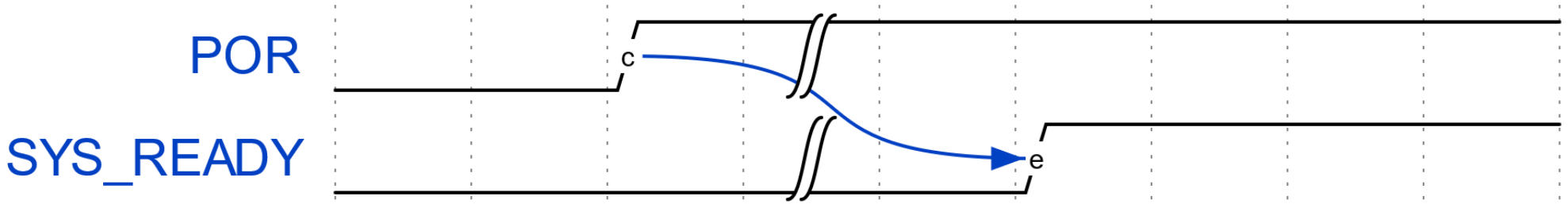



```
assert property ( @(posedge POR) 1 |-> @(posedge SYS_READY) 1 );
```

- This is a *weak* property
- Make it *strong*

```
assert property ( @(posedge POR) 1 |-> @(posedge SYS_READY) s_eventually(1) );
```

Coverage Alternative




```
bit coverage[string];
cover property ( @(posedge POR       ) 1) coverage["POR"       ]++;
cover property ( @(posedge SYS_READY) 1) coverage["SYS_READY"]++;
```

```
final begin
  if ( coverage["POR"] ) begin
     assert ( coverage["SYS_READY"] == coverage["POR"] ) else
        $error( ... );
  end
end
```

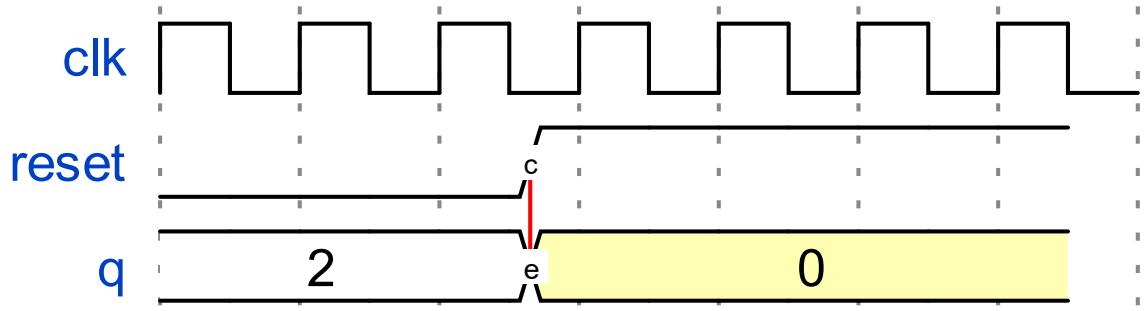
coverage[*]

POR	SYS_READY
1	1

Expect a cause for each effect

Scenario 2

Async signal causes RTL updates

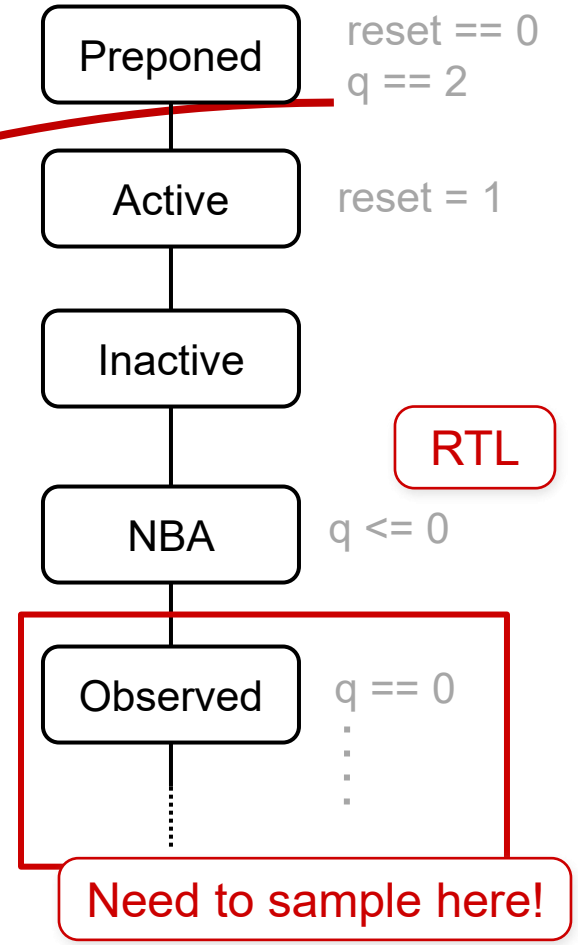



```
assert property ( @(posedge reset) q == 0 );
```

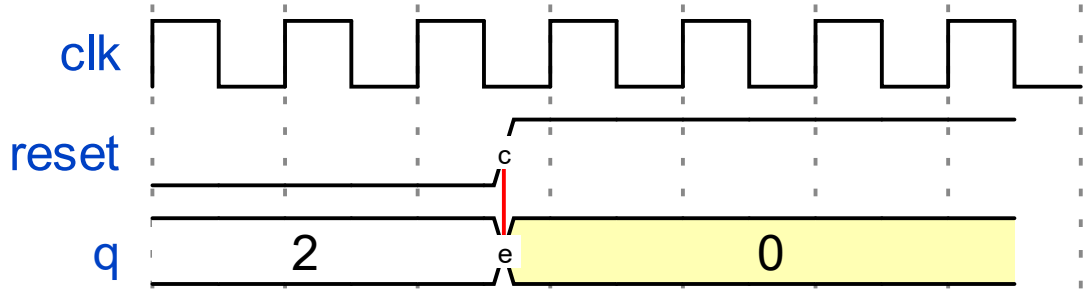
✗

Inputs from *Preponed*

q == 2



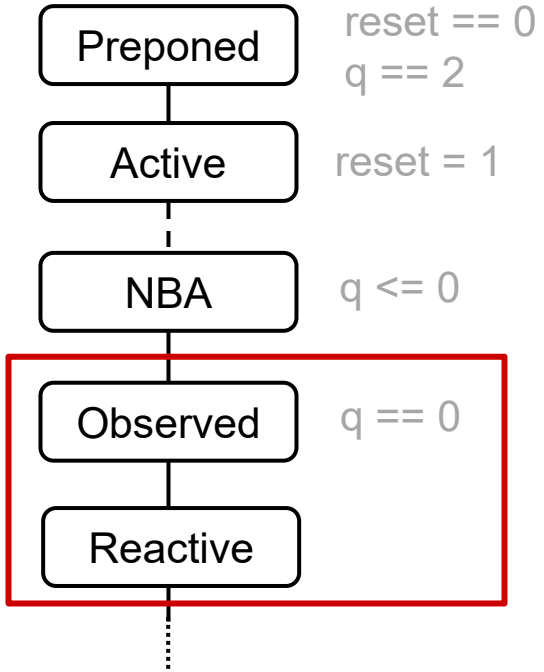
Program blocks



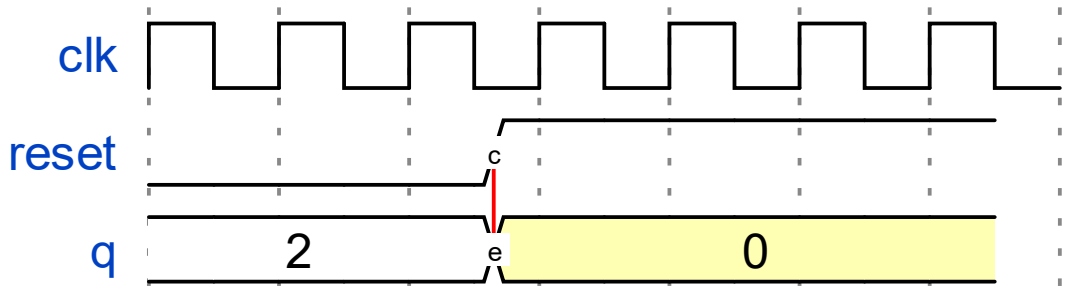
Inputs from *Observed*

```
program async_asserts;
   initial
      forever
        @(posedge reset)
            assert ( q == 0 );
endprogram
```

Scheduled in *Reactive*

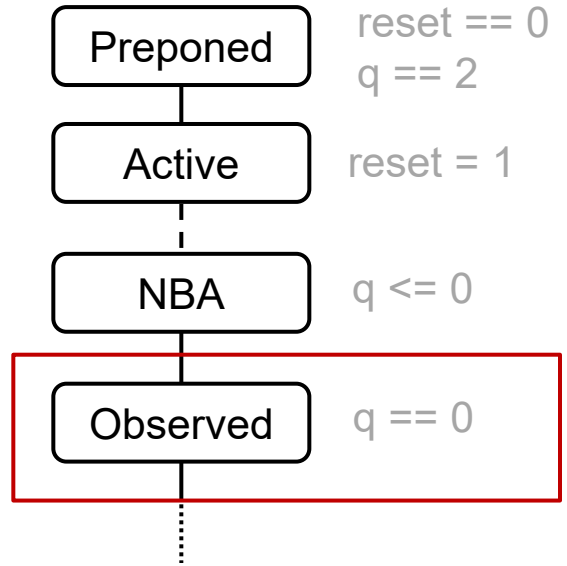


Sequence event

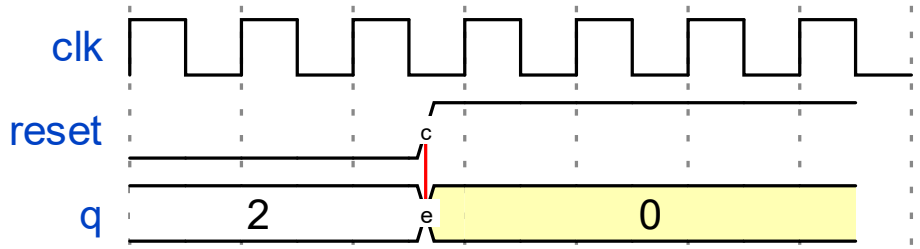



```
sequence seq_reset_event;
   @(posedge reset) 1;
endsequence

always
    @(seq_reset_event) assert ( q == 0 );
```

Sequence end point in *Observed*



Procedural concurrent assertions



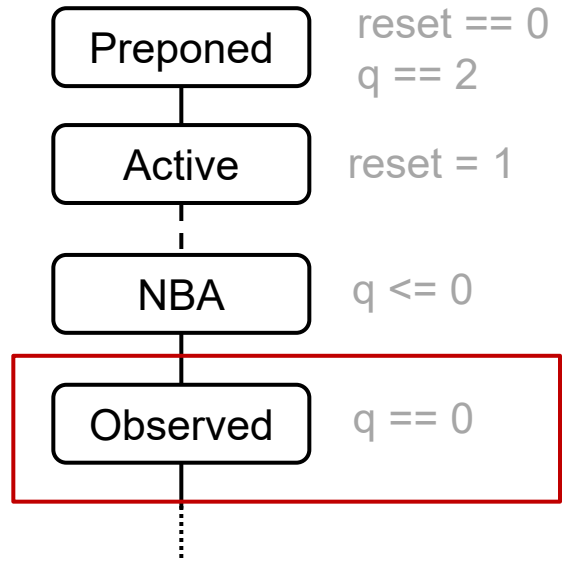
Procedural concurrent assertions mature in *Observed*

```
always @(posedge reset)
   assert property ( 1 )
      assert ( q == 0 );
```

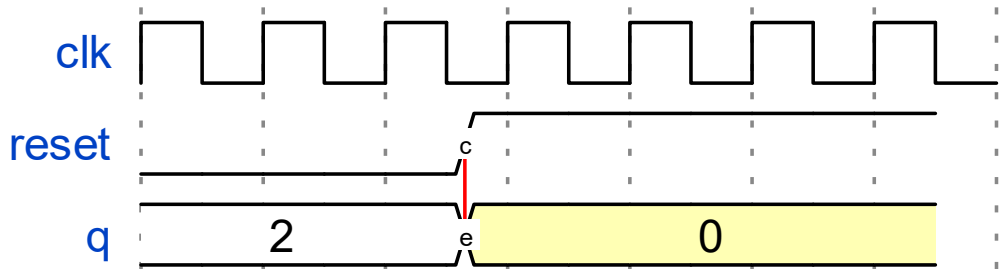
assert() executes in *Observed*

OR

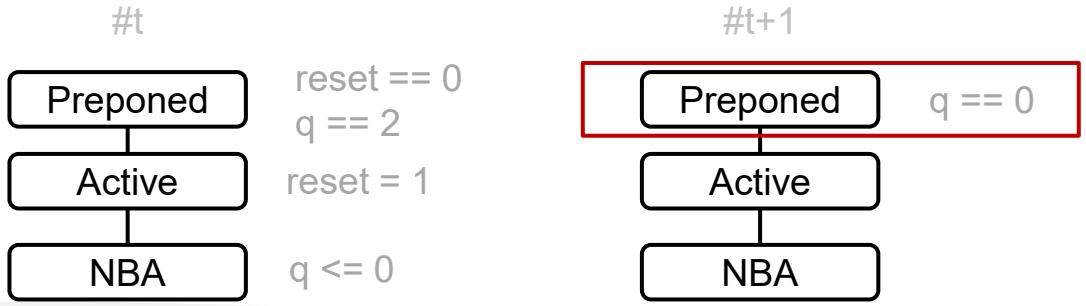
```
always_comb
   assert property ( @(posedge reset) 1 )
      assert ( q == 0 );
```



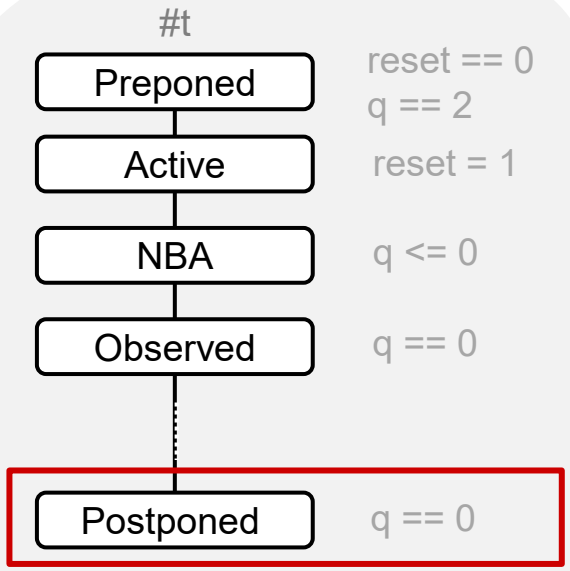
A timing delay




```
assign #1 delayed_reset = reset;
  assert property ( @(posedge delayed_reset) q == 0 );
```

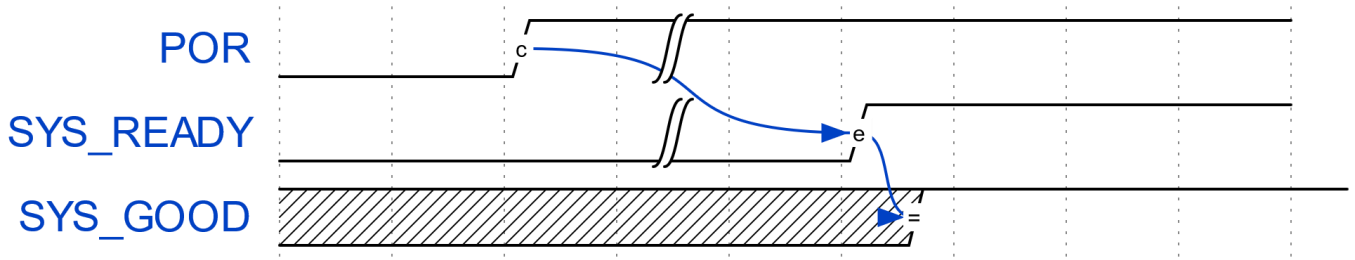



```
assign #1step delayed_reset = reset;
  assert property ( @(posedge delayed_reset) q == 0 );
```



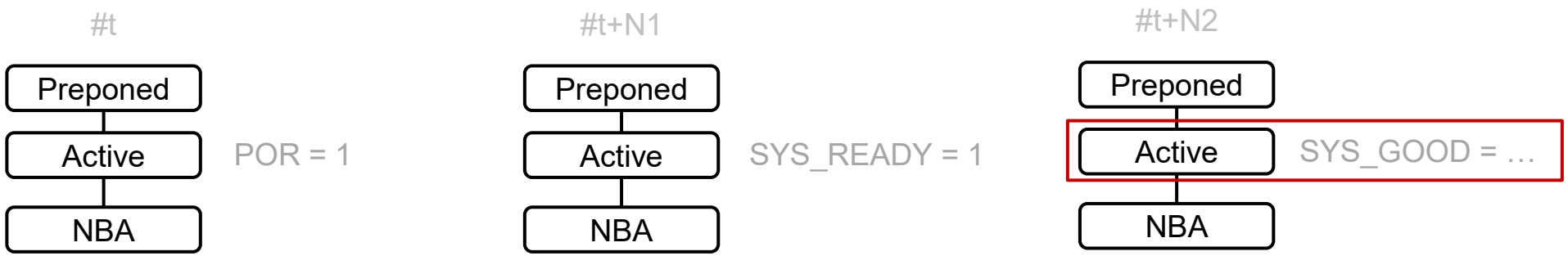
Scenario 3

Async event causes async event with updates

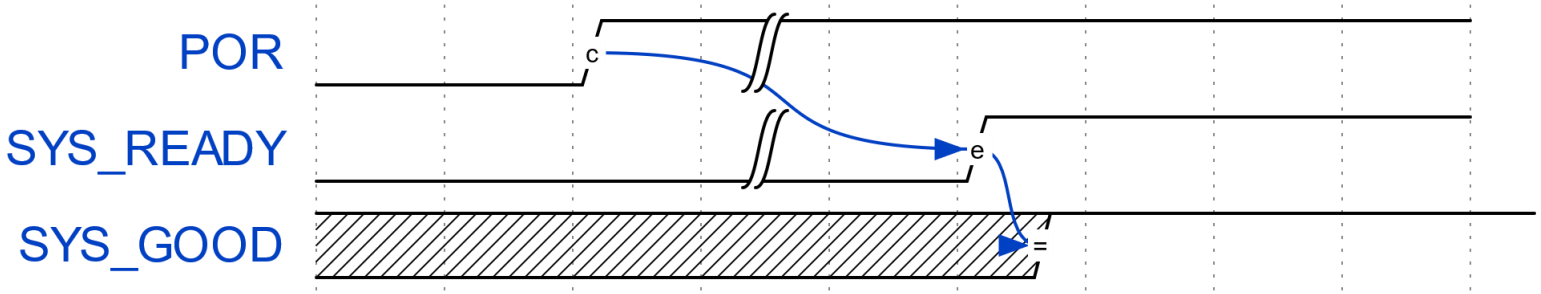


Multiclocked property

```
assert property (@(posedge POR) 1 |-> @(posedge SYS_READY) s_eventually(1)
                 |-> @(posedge SYS_GOOD) s_eventually(1));
```

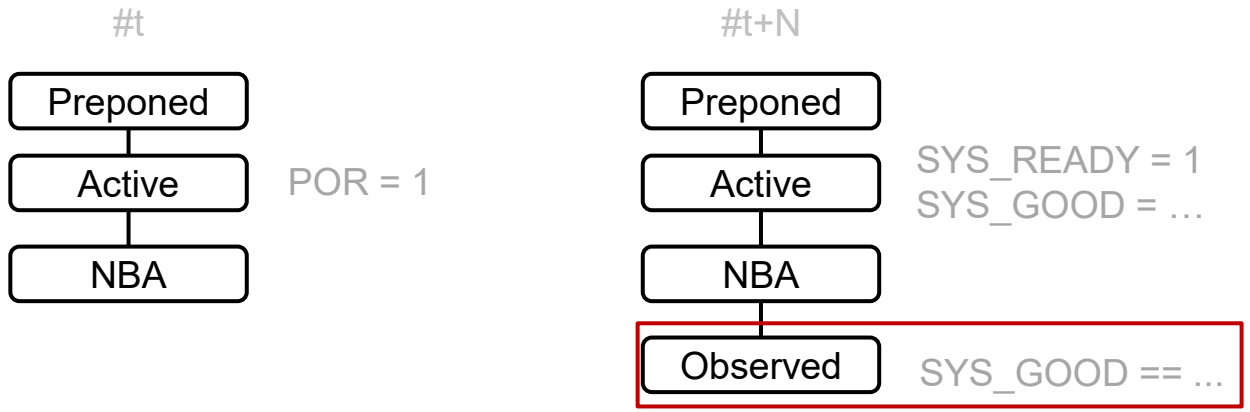


Overlapping behavior



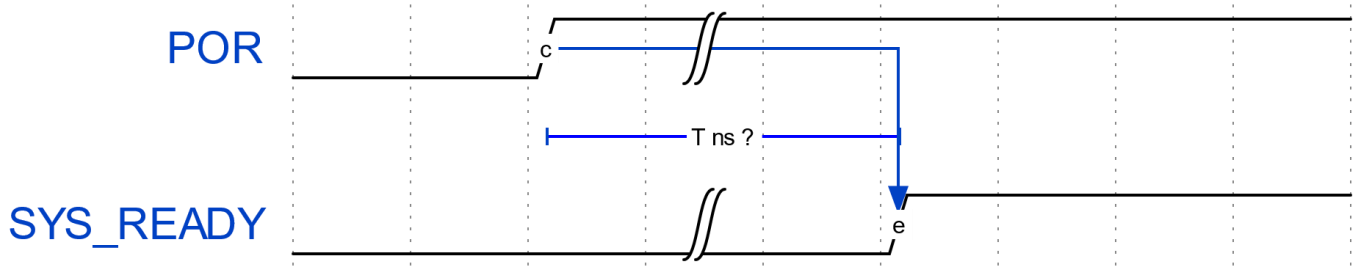

```
always_comb
  assert property ( @(posedge POR) 1 |-> @(posedge SYS_READY) s_eventually(1))
    assert(SYS_GOOD == ...);
```

assert() executes in *Observed*



Scenario 4

Async timing window

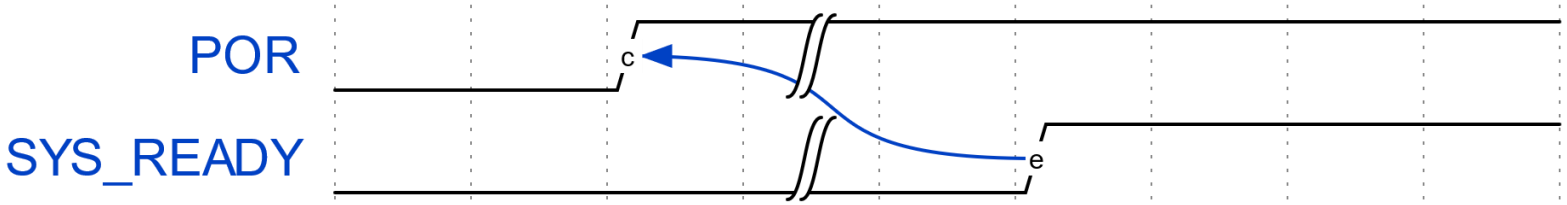



```
property prop_check_timing;
    realtime start;
    realtime finish;

    @(posedge POR)         (1, start  = $realtime) |->
    @(posedge SYS_READY)   (1, finish = $realtime) ##0
    (finish - start) == timing_window;
endproperty

assert property ( prop_check_timing );
```

The Effect and its Cause

Scenario 5

Async event caused by another event




```
bit cov_por;
cover  property ( @(posedge POR ) 1   ) cov_por = 1;
assert property ( @(posedge SYS_READY) cov_por );
```

OR

```
sequence seq_past_por;
    @(posedge POR) 1 ##1 @(posedge SYS_READY) 1;
endsequence

assert property ( @(posedge SYS_READY) seq_past_por.triggered );
```

Not as portable

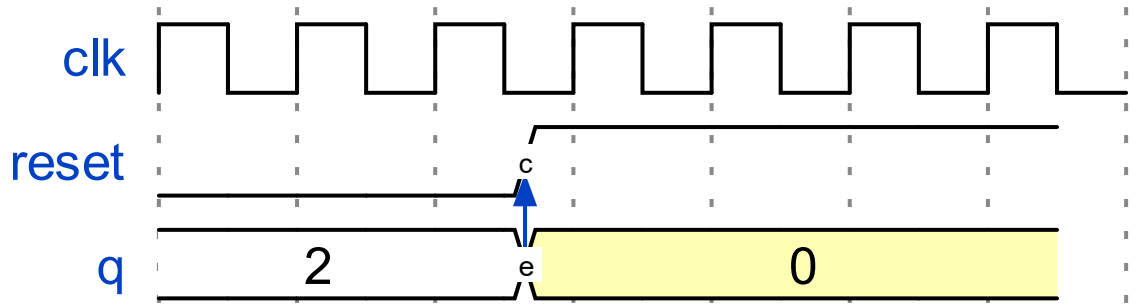
Overlapping effect-cause


```
sequence seq_past_por;
    @(posedge POR) 1 ##0 @(posedge SYS_READY) 1;
endsequence

assert property ( @(posedge SYS_READY) seq_past_por.triggered );
```

Avoid - inconsistent support across tools

Scenario 6

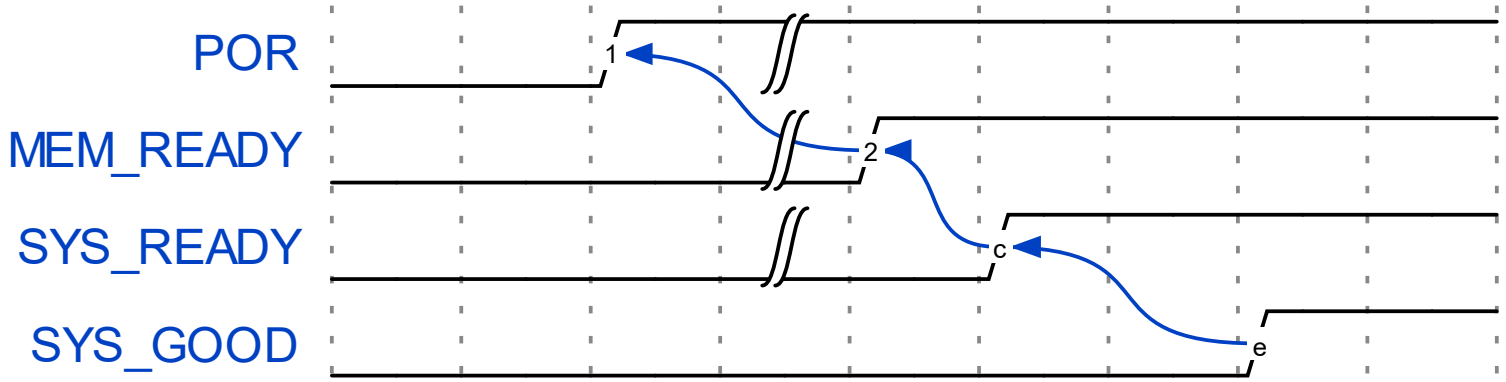
RTL updated by async event




```
bit cov_reset;
always @(posedge reset) cov_reset = 1;

always_comb
    assert property ( @(q) 1 )
       assert ( cov_reset )
          cov_reset = 0;
       else $error ( … );
```

Scenario 7

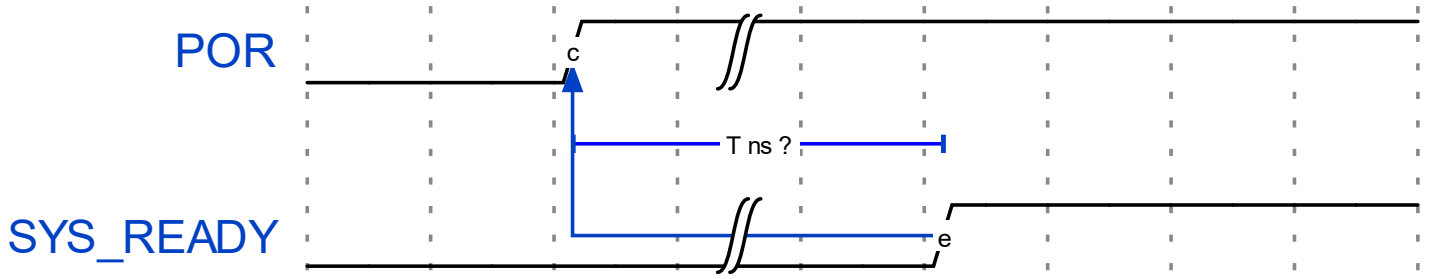
Multiple causes for a sequence of events




```
bit cov_por, cov_mem_ready, cov_sys_ready;
cover  property ( @(posedge POR)       1) cov_por = 1;
cover  property ( @(posedge MEM_READY) 1) cov_mem_ready = 1;
cover  property ( @(posedge SYS_READY) 1) cov_sys_ready = 1;

assert property ( @(posedge SYS_GOOD) cov_por &&
                                      cov_mem_ready &&
                                      cov_sys_ready );
```

Scenario 8

Async timing window effect-and-cause




```
bit cov_por;
realtime start;
cover property ( @(posedge POR) 1 ) begin
   cov_por = 1;
   start = $realtime;
end

assert property ( @(posedge SYS_READY) cov_por &&
                  (($realtime - start) <= timing_window ));
```

Asynchronous Communication

Asynchronous communication

Common types:

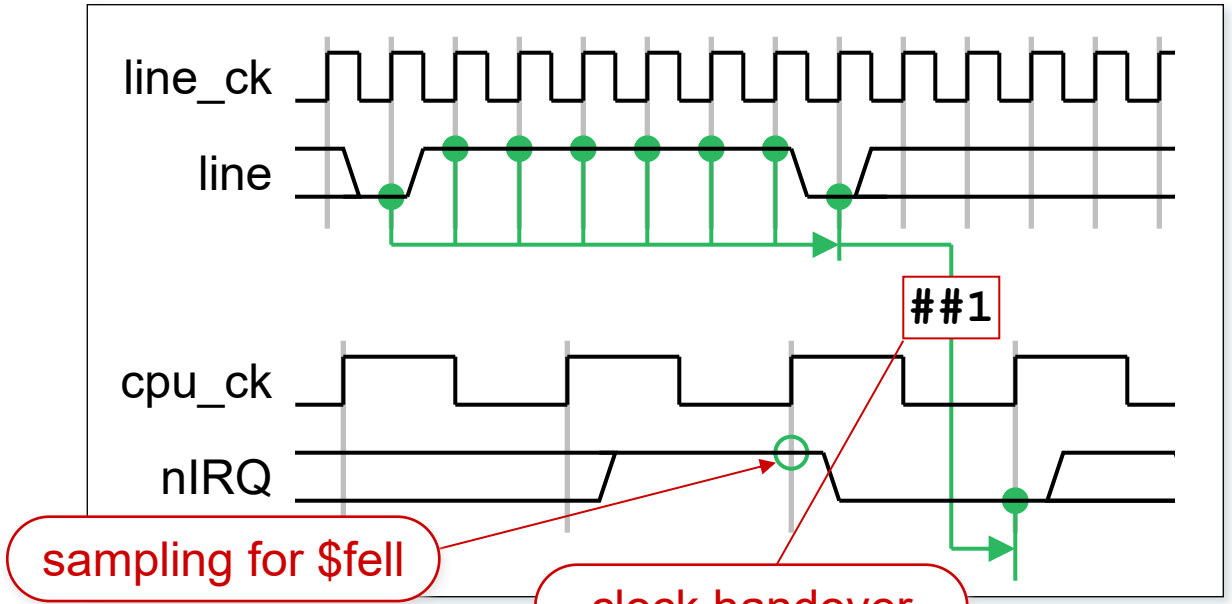
- Clock domain crossing
- Interface handshaking

Solution: *multi-clocked sequences*

Clock domain crossing

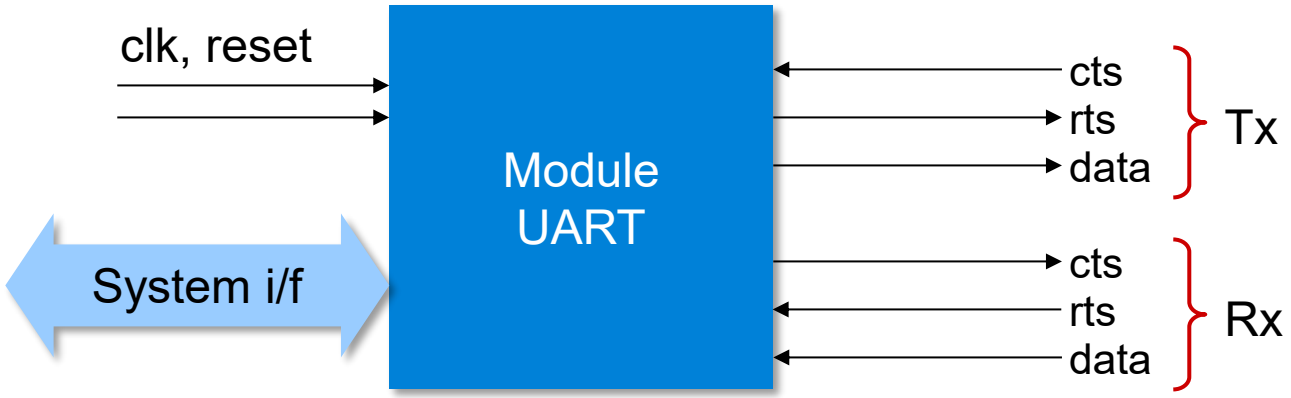
```
sequence flag; !line ##1 line[*6] ##1 !line; endsequence
sequence irq; $fell( nIRQ ); endsequence
```

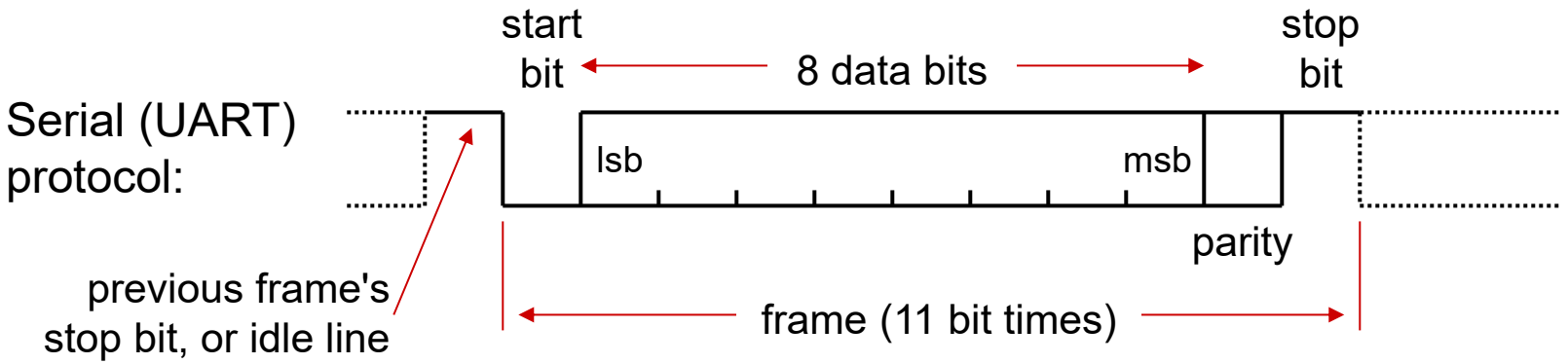
unclocked




```
assert property
   ( @(posedge line_ck) flag ##1 @(posedge cpu_ck) irq );
```

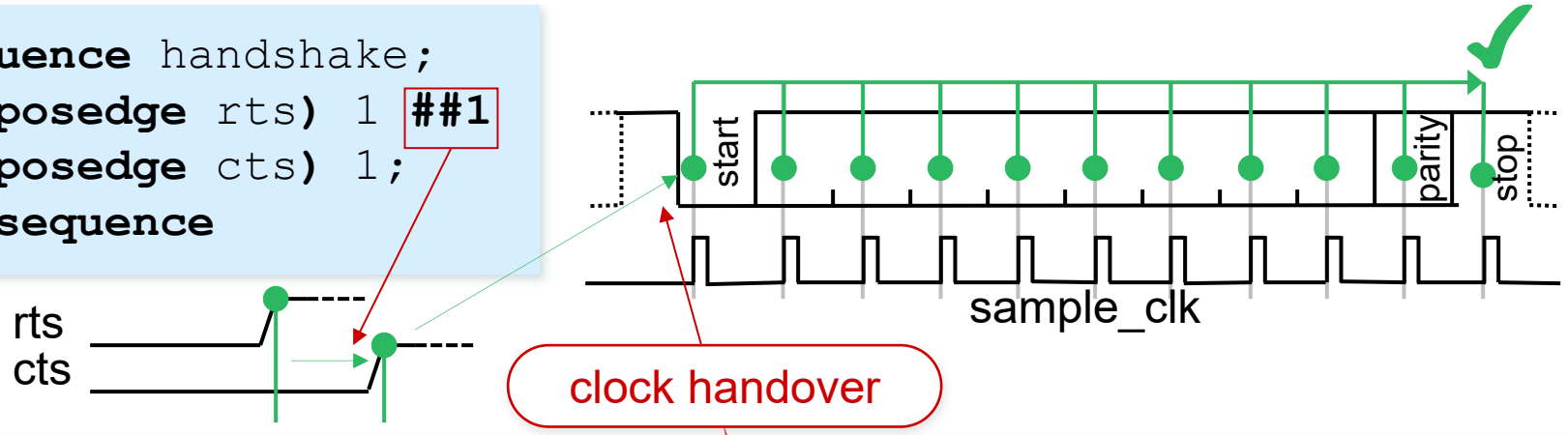
Asynchronous protocol - UART





Async transfer with sampling

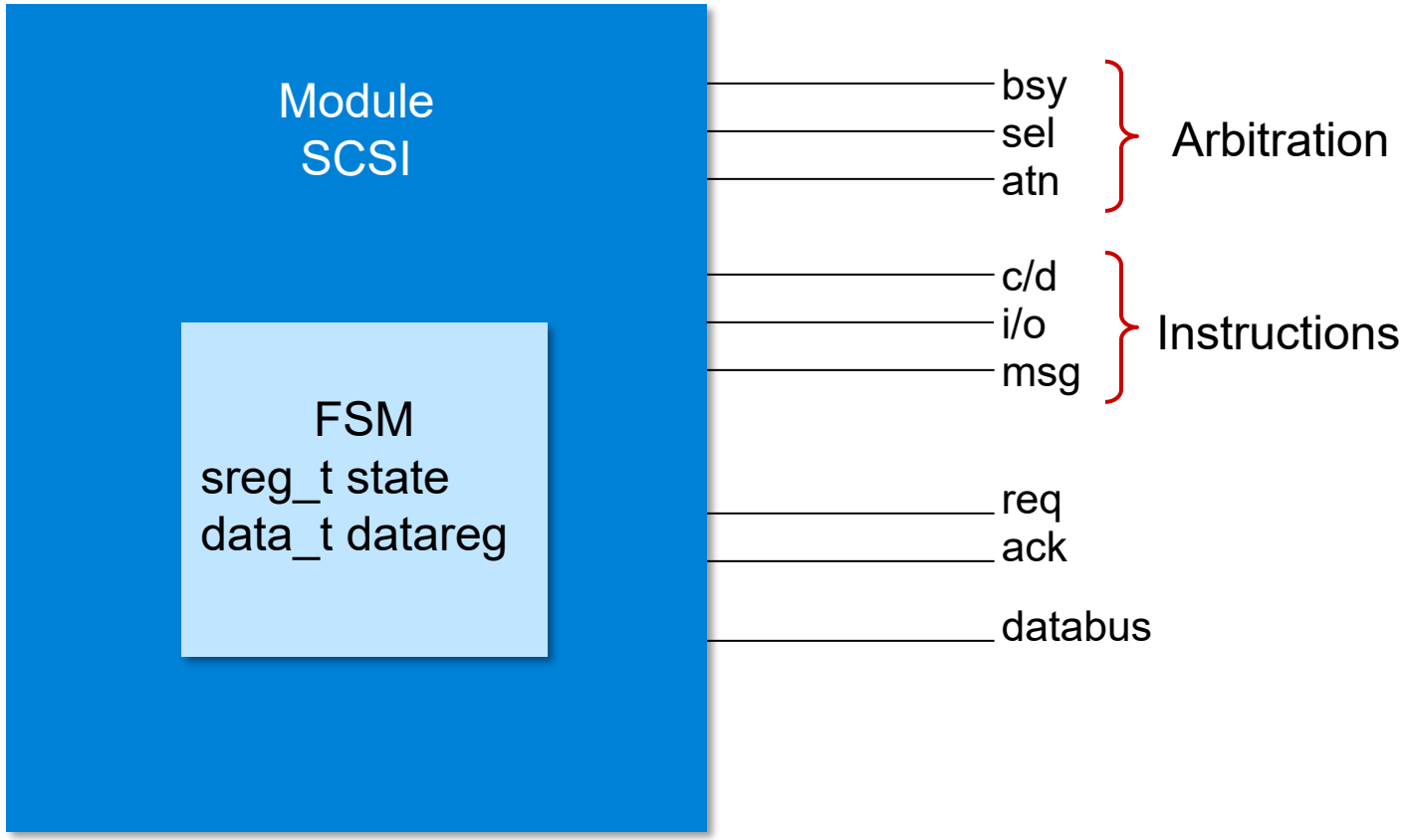
```
sequence handshake;
  @(posedge rts) 1 ##1
  @(posedge cts) 1;
endsequence
```



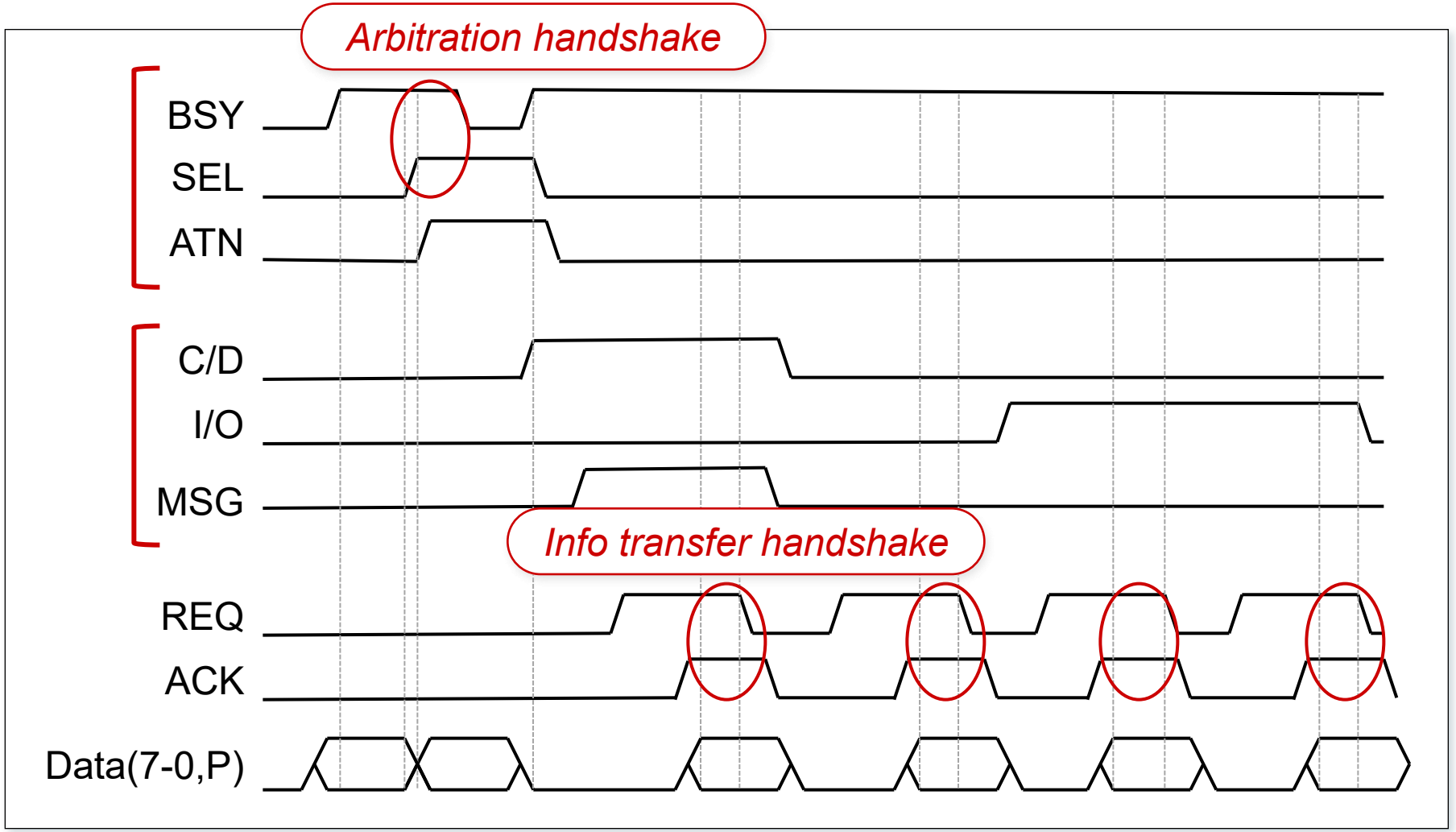

```
assert property ( handshake |=> check_trans );
```

```
sequence check_trans;
  logic [7:0] tr;                        // Local variable
  @(posedge sample_clk)  1     ##1       // Skip start bit
            (1, tr[0] = data) ##1 (1, tr[1] = data) ##1
            (1, tr[2] = data) ##1 (1, tr[3] = data) ##1
            (1, tr[4] = data) ##1 (1, tr[5] = data) ##1
            (1, tr[6] = data) ##1 (1, tr[7] = data) ##1
           data === ^tr       ##1       // Check parity
           data === 1;                   // Check stop bit
endsequence
```

Simplified SCSI I/O



SCSI protocol

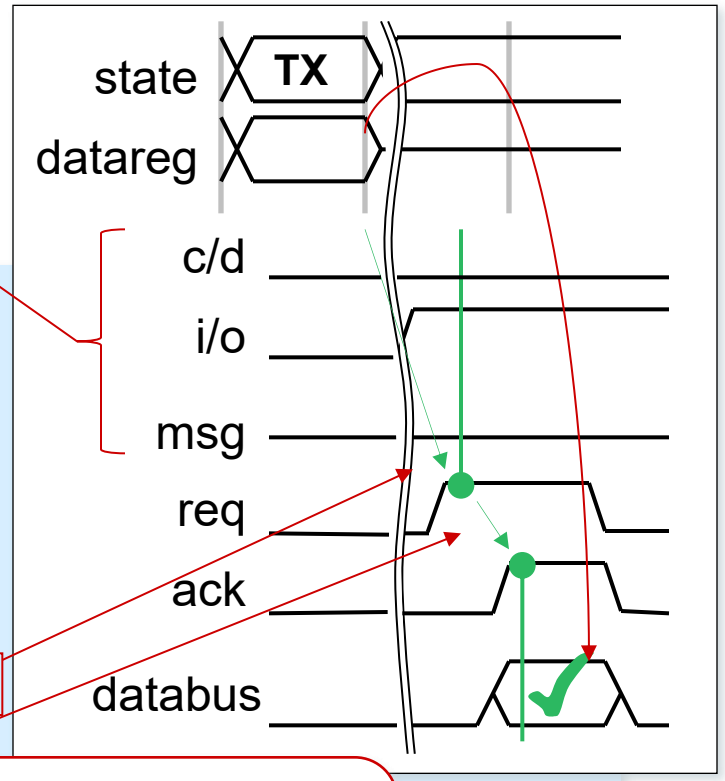


Handshaking assertion

```
sequence data_cmd;
  !cd && io && !msg;
endsequence
```

```
property check_data;
 data_t txdata;   // Local variable

 @(posedge clk)
 ( state == TX, txdata = datareg ) |=>
       @(posedge REQ) data_cmd ##1

       @(posedge ACK) databus == txdata;
endproperty
assert property ( check_data );
```

RTL signals

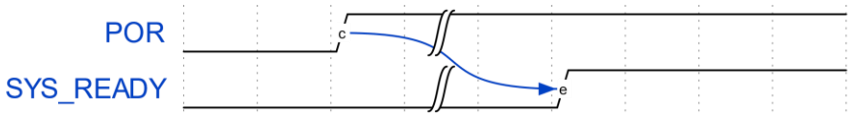


clock handover

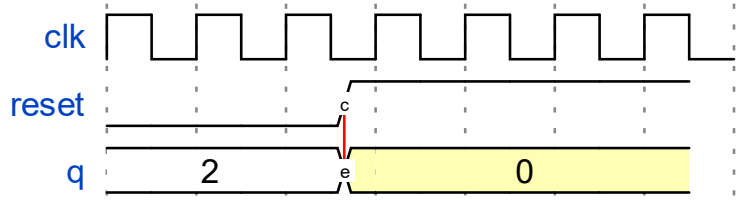
Summary

Cause and effect scenarios

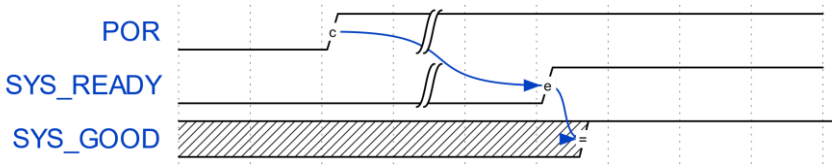
(1) Async signal causes another async event



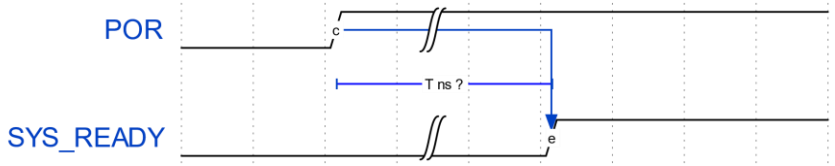
(2) Async signal causes RTL updates



(3) Async event causes async event with updates

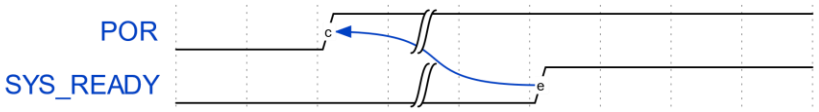


(4) Async timing window

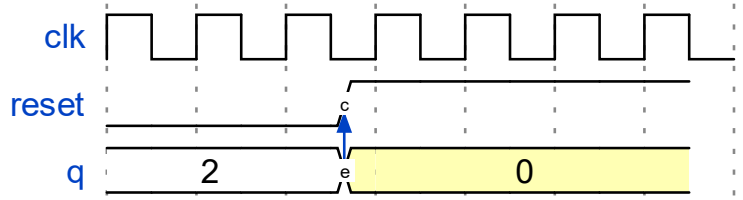


The effect and the cause scenarios

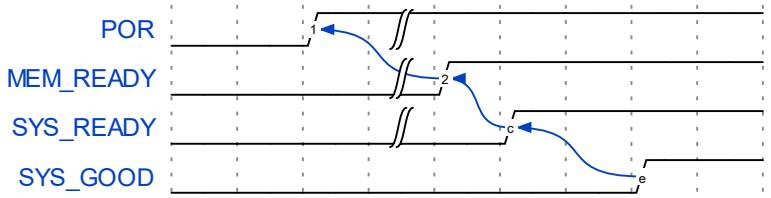
(5) Async event caused by another event



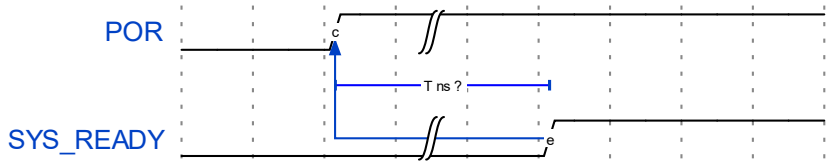
(6) RTL updated by async event



(7) Multiple causes for a sequence of events



(8) Async timing window effect-and-cause



Recommendations

Asynchronous bus protocols

- Use multi-clocked properties (usually straightforward)

Asynchronous controls

- Oversampling generally good enough
- Coverage approach works in most cases (plus bonus of functional coverage)
- Other scenarios, find a way to delay the checker’s sampling

DOULOS

Delivering KnowHow www.doulos.com

SoC Design & Verification	» SystemVerilog » UVM » Formal » SystemC » TLM-2.0
FPGA & Hardware Design	» VHDL » Verilog » SystemVerilog » Tcl » AMD
Embedded Software & Arm	» Emb C/C++ » Emb Linux » Yocto » RTOS » Security » Android » Arm » Rust » Zephyr
Python, AI & Machine Learning	» Python » Edge AI » Deep Learning

Examples available at: https://edaplayground.com/x/qB72

SystemVerilog OUT OF THE BOX

UVM

SYSTEMC

yocto PROJECT PARTICIPANT

Zephyr®

SECURE EMBEDDED

arm APPROVED TRAINING PARTNER

AI & MACHINE LEARNING

New Design and Verification courses

Advanced Formal Verification

- equips you to tackle complex verification challenges by taking full advantage of formal verification in your engineering projects.
- forms a complete learning path with **Essential Formal Verification** course, which gives you a solid, practical grounding in formal verification.

SystemVerilog for New Designers: Self-Paced course

- get project-ready for FPGA or ASIC design, including RTL synthesis, block-level test benches, and FPGA design flows.
- high-quality content developed by expert instructors - now in self-paced format.

IC Verification with Python and cocotb

- learn cocotb principles, constructing different aspects of a verification environment using cocotb and Verification strategies and tactics.


Questions?