Applying Stimulus & Sampling Outputs UVM Verification Testing Techniques

Clifford E. Cummings

Sunburst Design, Inc.

World-class SystemVerilog & UVM Verification Training

Life is too short for bad or boring training!

September 29, 2016

SNUG Austin

Free IEEE SystemVerilog-2012 LRM @
`http://standards.ieee.org/getieee/1800/download/1800-2012.pdf`

Agenda

Time-0 race conditions

Stimulus & verification goals

Driving stimulus - timing

Sampling inputs for prediction model

Sampling outputs - verification timing

SystemVerilog `program`

Summary & conclusions

3 important timing points for testbenches and vectors

The paper has more details and more examples

Recommended reading: Bromley & Johnston's SNUG-Austin 2012 detailed paper on Clocking Blocks

Time-0 Simulation and Race Conditions

How to Avoid Time-0 Race Conditions

Time-0

Race conditions

- Time-0 is a tricky place in Verilog & SystemVerilog simulations
- At time-0 all `initial` and `always` blocks become active *in any order!*

The problems and solutions are on the next slides

Time-0 Race Conditions

initial - time-0 blocking assignments

```
module initial_always1;
  initial
    @(negedge clk) $display("%t: initial #1 negedge clk",$time);

  always begin
    @(negedge clk) $display("%t: always #1 negedge clk",$time);
    wait(0);
  end

  initial begin
    clk = '0;
    forever #(`CYCLE/2) clk = ~clk;
  end

  initial
    @(negedge clk) $display("%t: initial #2 negedge clk",$time);

  always begin
    @(negedge clk) $display("%t: always #2 negedge clk", $time);
    wait(0);
  end   ...
```

Active at time-0

Active at time-0

Time-0 blocking assignment

Active at time-0

Active at time-0

VCS output

```
 0ns: initial #1 negedge clk
 0ns: always #1 negedge clk
 0ns: always #2 negedge clk
10ns: initial #2 negedge clk
25ns: FINISH
```

Simulator B output

```
 0ns: always #1 negedge clk
 0ns: initial #1 negedge clk
10ns: always #2 negedge clk
10ns: initial #2 negedge clk
25ns: FINISH
```

Both outputs are correct - Time-0 race condition!

Time-0 Race Conditions

initial - time-0 ***nonblocking*** assignments

```
module initial_always2;
  initial
    @(negedge clk) $display("%t: initial #1 negedge clk",$time);

  always begin
    @(negedge clk) $display("%t: always #1 negedge clk",$time);
    wait(0);
  end

  initial begin
    clk <= '0;
    forever #(`CYCLE/2) clk = ~clk;
  end

  initial
    @(negedge clk) $display("%t: initial #2 negedge clk",$time);

  always begin
    @(negedge clk) $display("%t: always #2 negedge clk", $time);
    wait(0);
  end   ...
```

Active at time-0

Active at time-0

Time-0 *nonblocking* assignment

Active at time-0

Active at time-0

VCS output

```
  0ns: initial #1 negedge clk
  0ns: always #1 negedge clk
  0ns: initial #2 negedge clk
  0ns: always #2 negedge clk
 15ns: FINISH
```

Simulator B output

```
  0ns: always #2 negedge clk
  0ns: initial #2 negedge clk
  0ns: always #1 negedge clk
  0ns: initial #1 negedge clk
 15ns: FINISH
```

All blocks were active before first `negedge clk` - *NO* time-0 race condition!

`initial` - bad name! should have been called `run_once`

Time-0

How to handle time-0 stimulus

- To avoid time-0 race conditions
  - Make all time-0 stimulus assignments using nonblocking assignments
  - Create and call an `initialize()` task for time-0 stimulus assignments

UVM example
(explained in detail on a later slide)

```
  ...

task initialize();
  `uvm_info("RESET", "Initial reset", UVM_MEDIUM)
  vif.rst_n <= '0;
  vif.ld    <= '1;
  vif.inc   <= '1;
  vif.din   <= '1;
endtask


task drive_tr (trans1 tr);
  @(vif.cb1);
  vif.cb1.din   <= tr.din; ...
endtask
endclass
```

The `initialize()` task is called at time-0 and uses nonblocking assignments

```
task run_phase(uvm_phase phase);
  trans1 tr;
  initialize();
  forever begin
    seq_item_port.get_next_item(tr);
    drive_tr(tr);
    seq_item_port.item_done();
  end
endtask
```

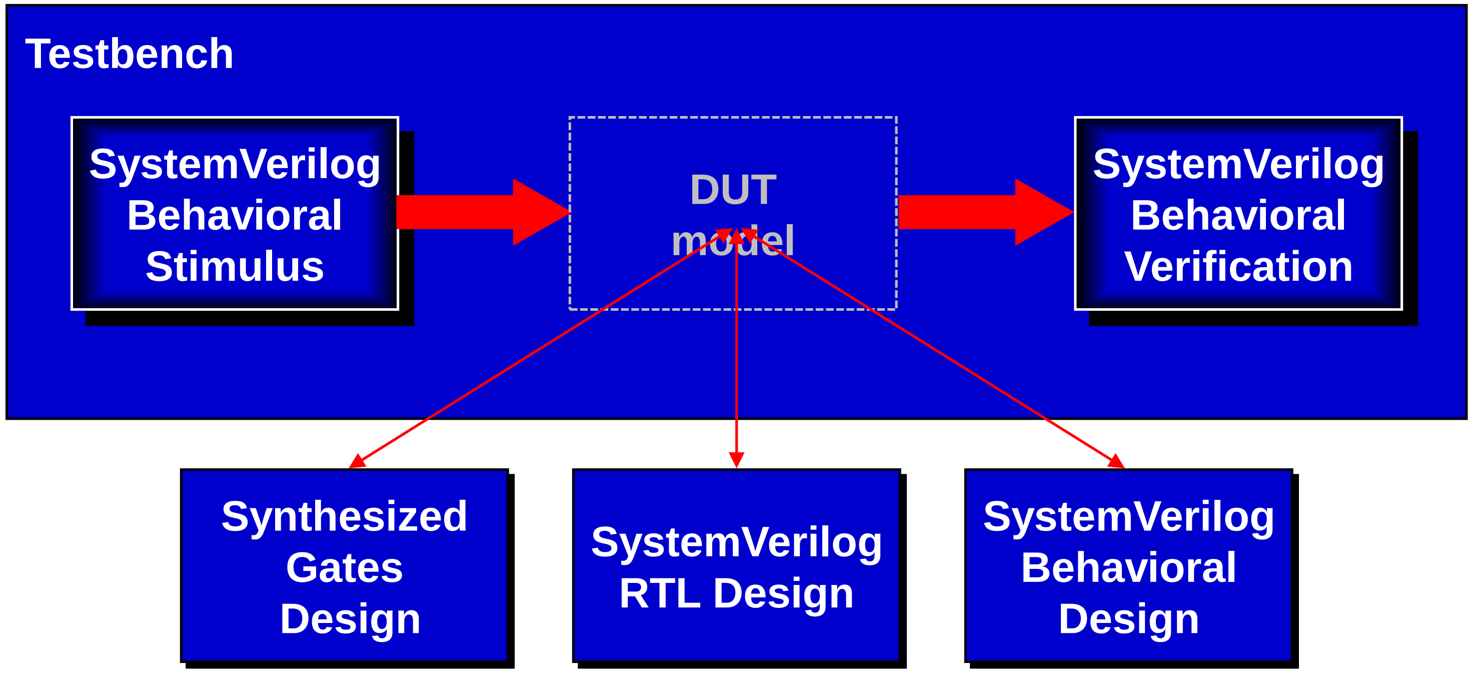
`initialize()` violates Bromley & Johnston Guideline #1 *but* time-0 is an important exception

The `drive_tr` task will use a `clocking` block to control stimulus timing *(explained on later slide)*

Stimulus and Verification Goals

Stimulus and Verification Vectors

- When and how should the stimulus be applied?
- When and how should the outputs be verified?


Testbench
SystemVerilog Behavioral Stimulus
DUT model
SystemVerilog Behavioral Verification
Synthesized Gates Design
SystemVerilog RTL Design
SystemVerilog Behavioral Design

We want a common testbench for all phases of the design

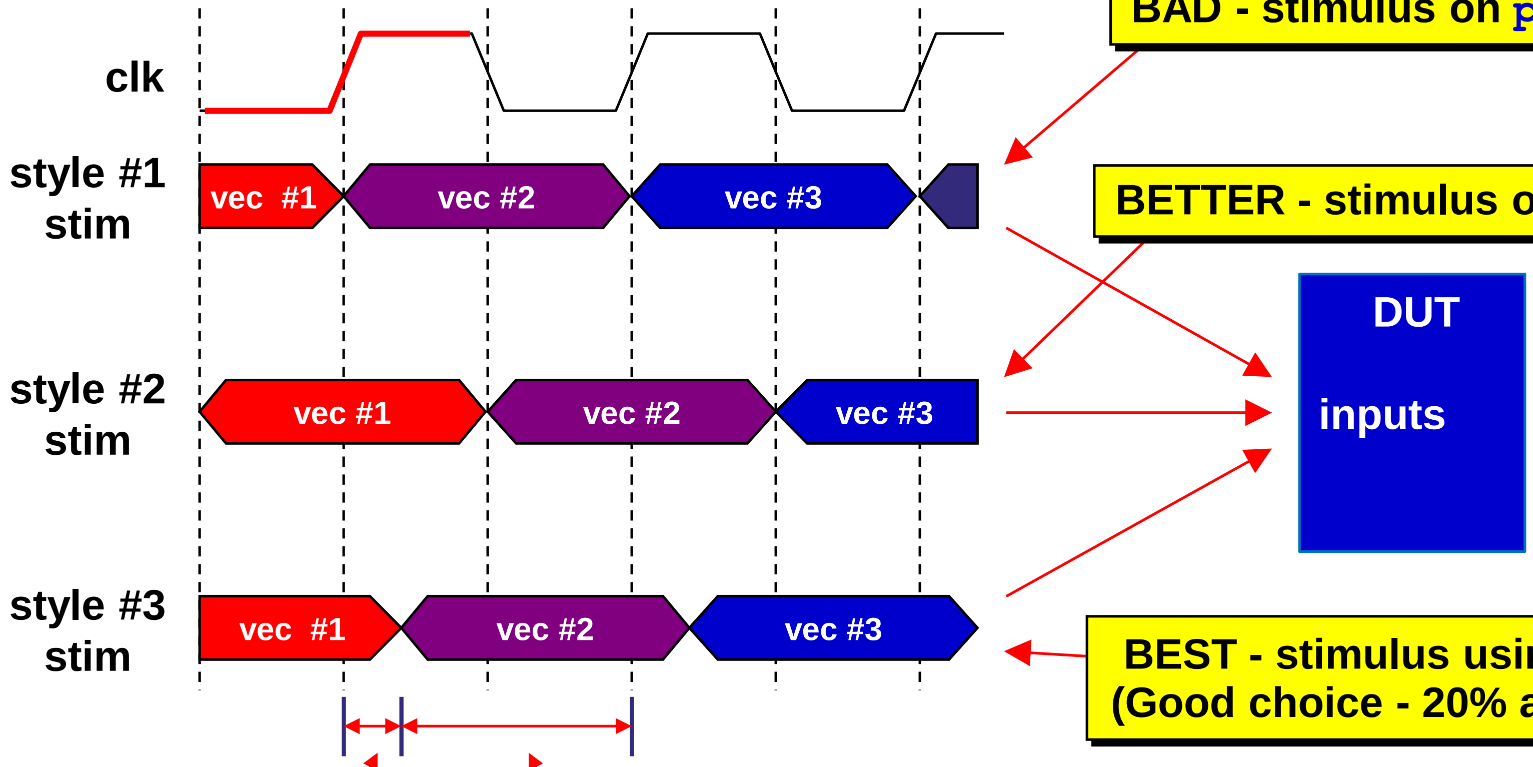
We will examine:
- sequential logic verification timing

Stimulus Strategies & Timing

Stimulus Driving Strategies

Bad, Better, Best

Preview of three common stimulus strategies



BAD - stimulus on `posedge clk`

BETTER - stimulus on `negedge clk`

BEST - stimulus using time budgeting (Good choice - 20% after `posedge clk`)

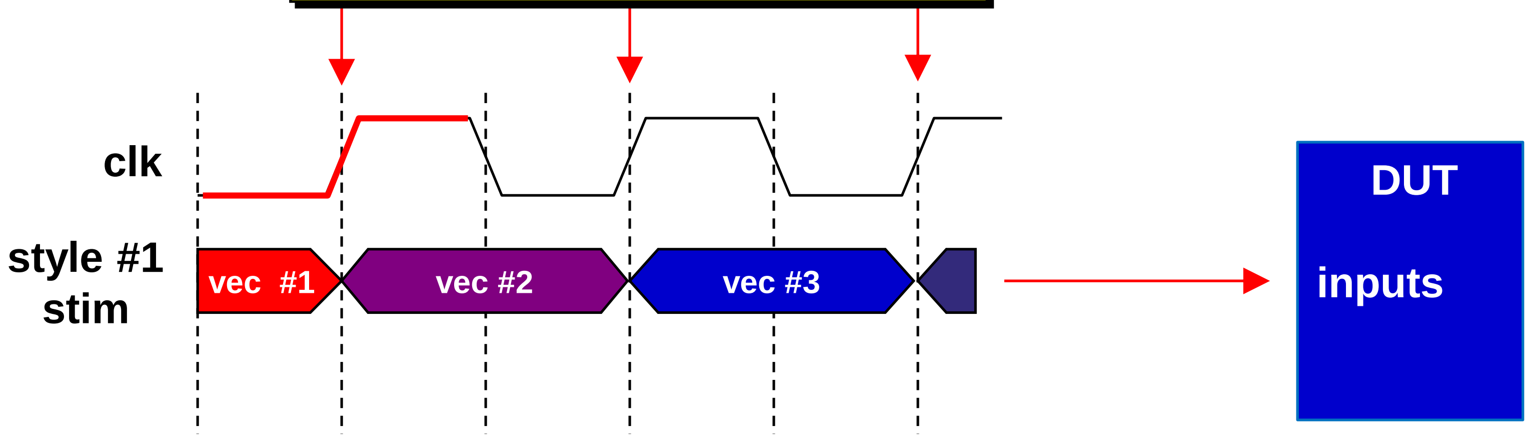
20% `clk`-to-`q` time budget

80% combinational input time budget

Stimulus Driving Strategy

Active Clock Stimulus

BAD - stimulus on `posedge clk`

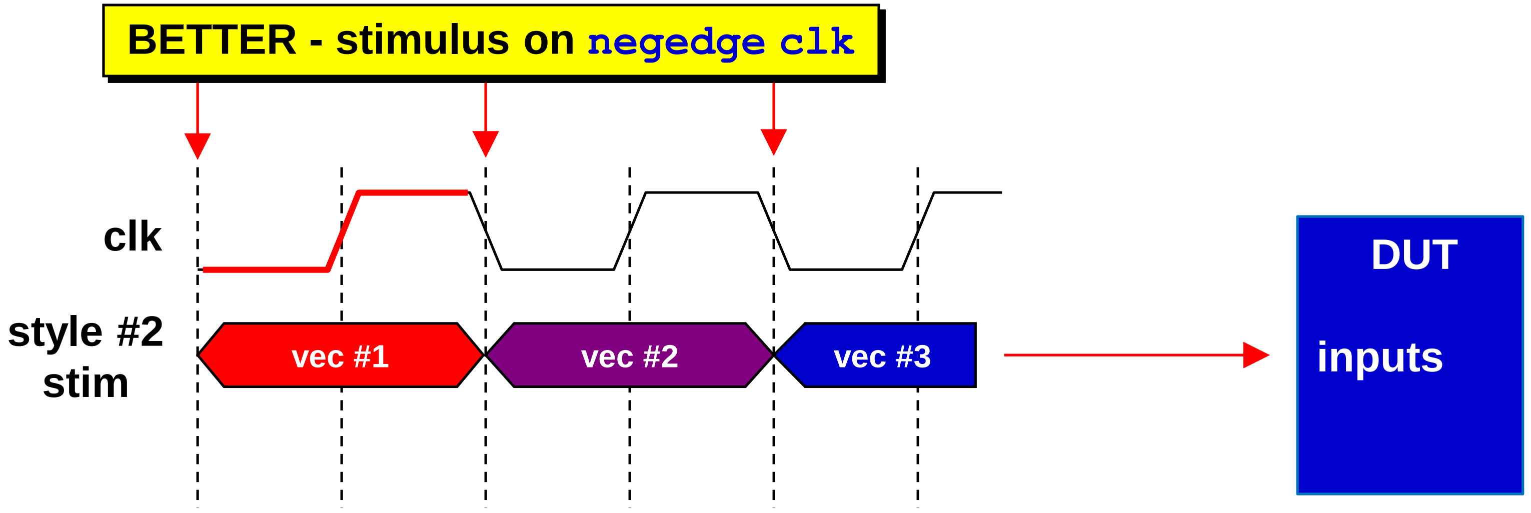


Advantage
(1) *Should* work for 0-delay RTL Designs

Disadvantages
(1) Requires nonblocking assignments to drive stimulus
(2) Fails setup/hold times for gate-sims
(3) Requires testbench modifications to do gate-sims

Stimulus Driving Strategy

Inactive Clock Stimulus



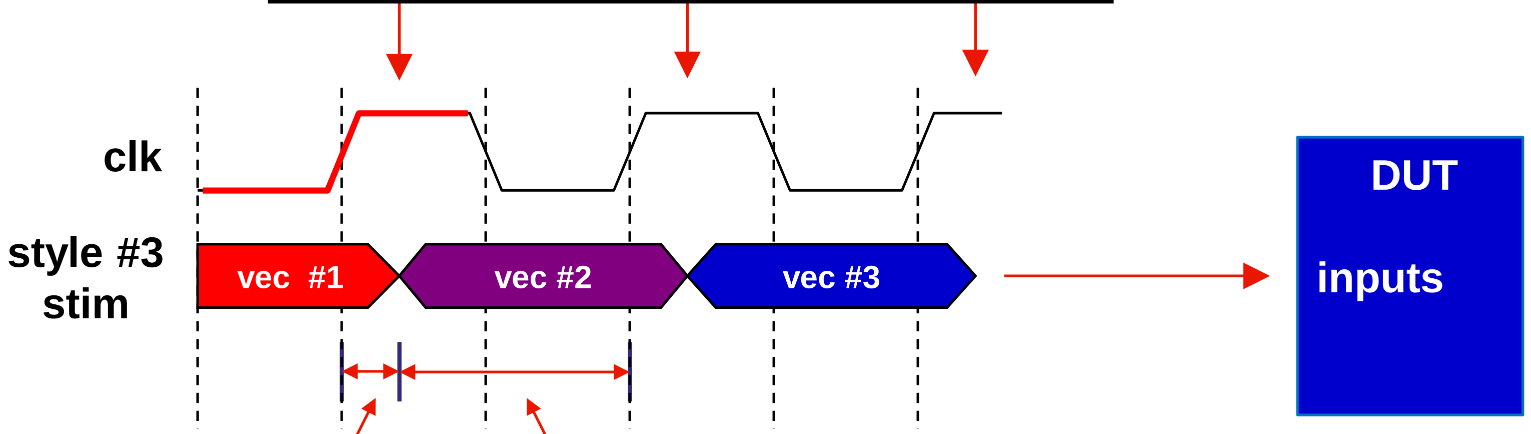
Advantage
(1) Works for 0-delay RTL Designs and most gate-sims

Disadvantage
(1) Fails gate-sims if DUT input combinational delay exceeds ½ cycle

Stimulus Driving Strategy

Time-Budget Stimulus

BEST - stimulus using time budgeting (Good choice - 20% after `posedge clk`)



20% `clk` -to- `q` time budget

80% combinational input time budget

Advantage

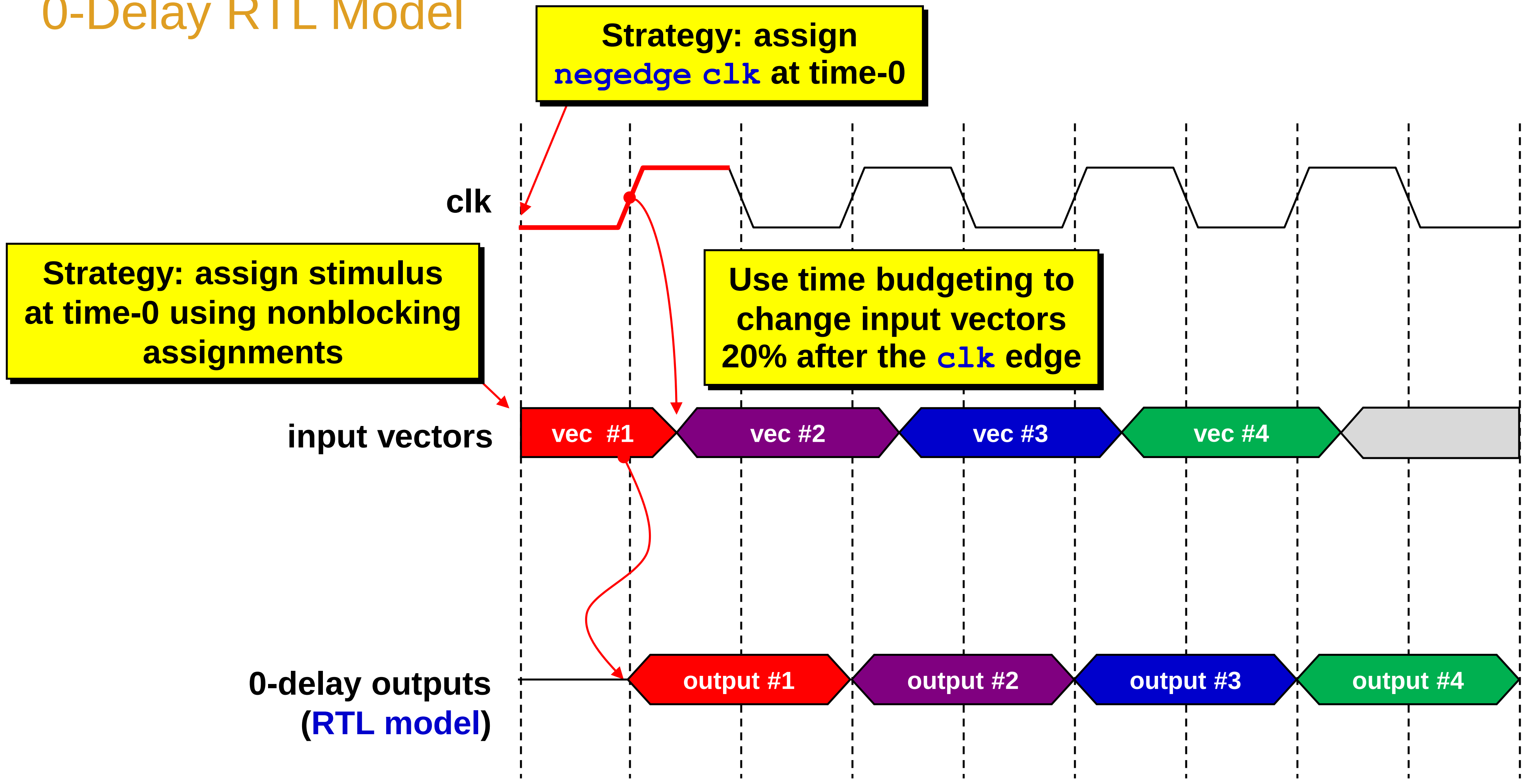
(1) Choose appropriate `clk`-to-`q` time budget to meet input combinational delay

(2) Works for 0-delay RTL & gates sims with delays

Verification Strategies & Timing

Sequential Logic Testing

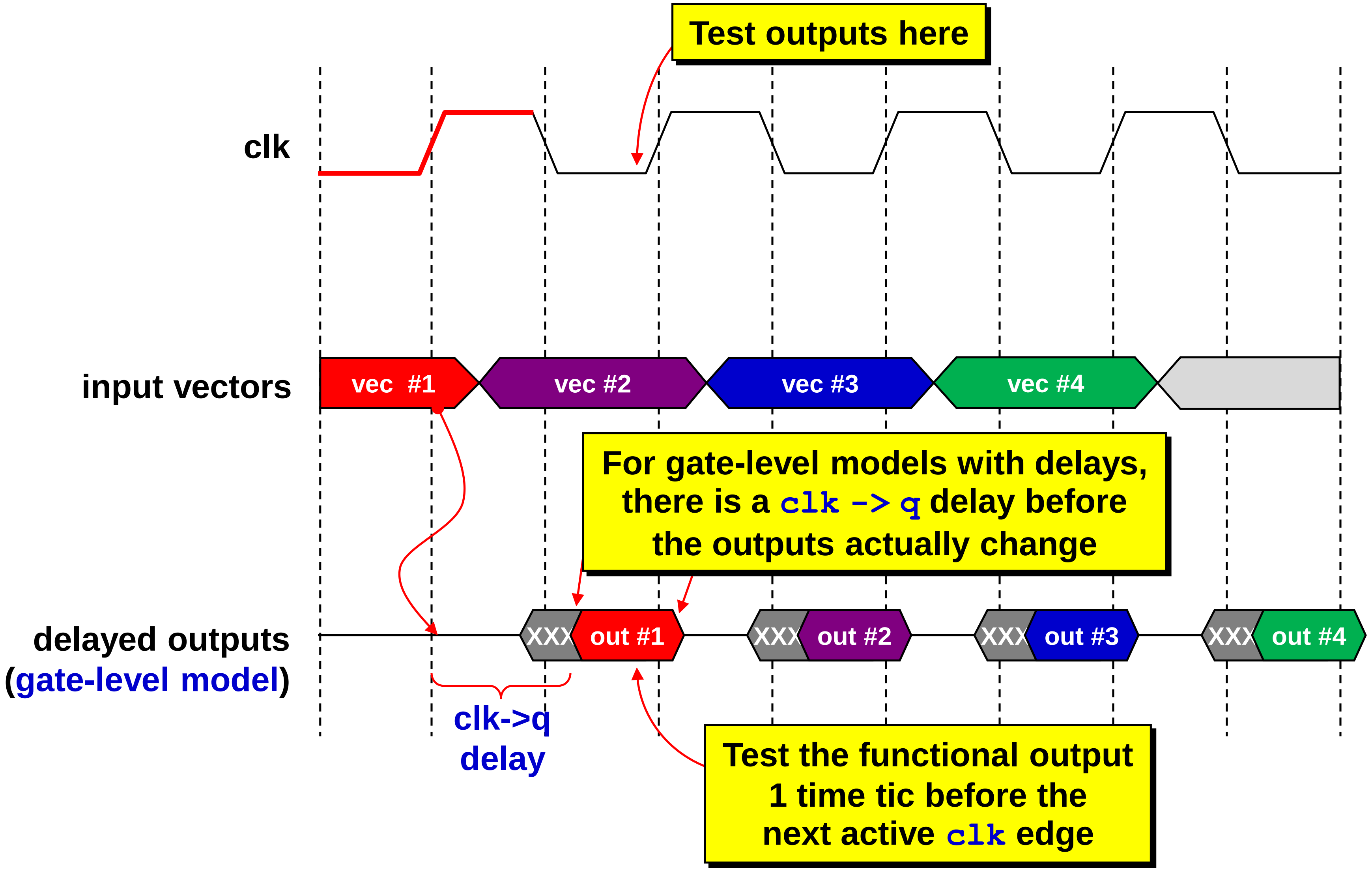
0-Delay RTL Model



Strategy: assign negedge clk at time-0
clk
Strategy: assign stimulus at time-0 using nonblocking assignments
Use time budgeting to change input vectors 20% after the clk edge
input vectors
vec #1
vec #2
vec #3
vec #4
0-delay outputs (RTL model)
output #1
output #2
output #3
output #4

Sequential Logic Testing

Gate-Level Model with Delays

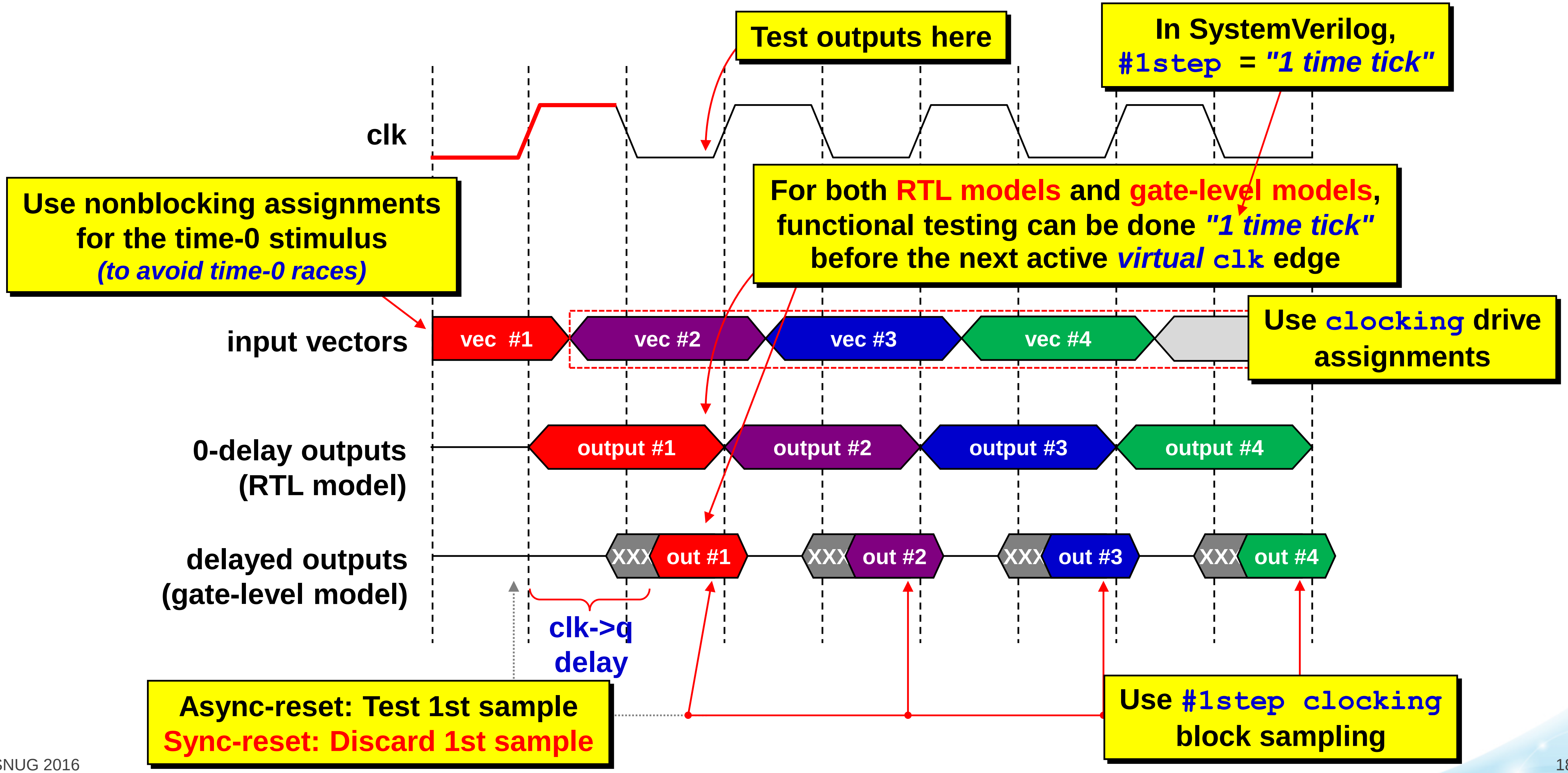


For gate-level models with delays, there is a `clk -> q` delay before the outputs actually change

Test the functional output 1 time tic before the next active `clk` edge

Sequential Logic Testing

Same Testing for RTL & Gates



Test outputs here
In SystemVerilog, #1step = "1 time tick"
clk
Use nonblocking assignments for the time-0 stimulus (to avoid time-0 races)
For both RTL models and gate-level models, functional testing can be done "1 time tick" before the next active virtual clk edge
input vectors
vec #1
vec #2
vec #3
vec #4
Use clocking drive assignments
0-delay outputs (RTL model)
output #1
output #2
output #3
output #4
delayed outputs (gate-level model)
XXX
out #1
out #2
out #3
out #4
clk->q delay
Async-reset: Test 1st sample
Sync-reset: Discard 1st sample
Use #1step clocking block sampling

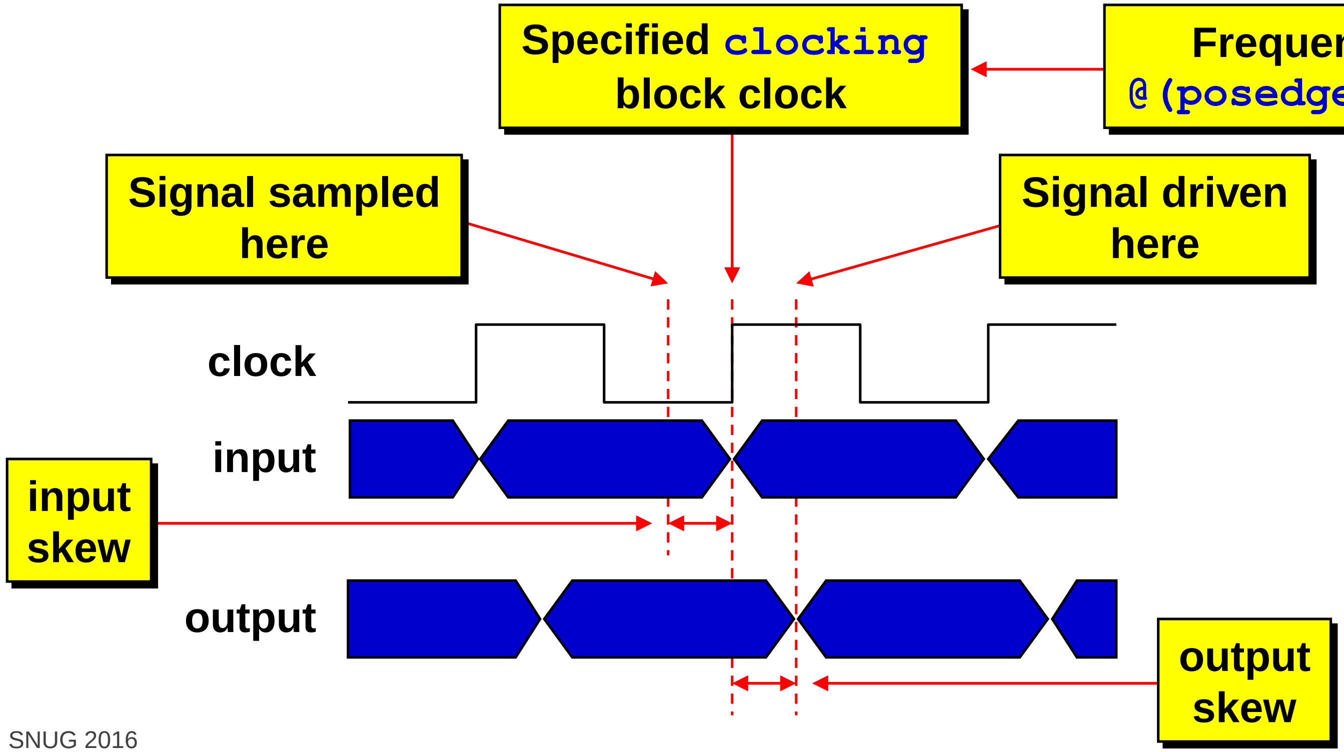
Clocking Block Skews

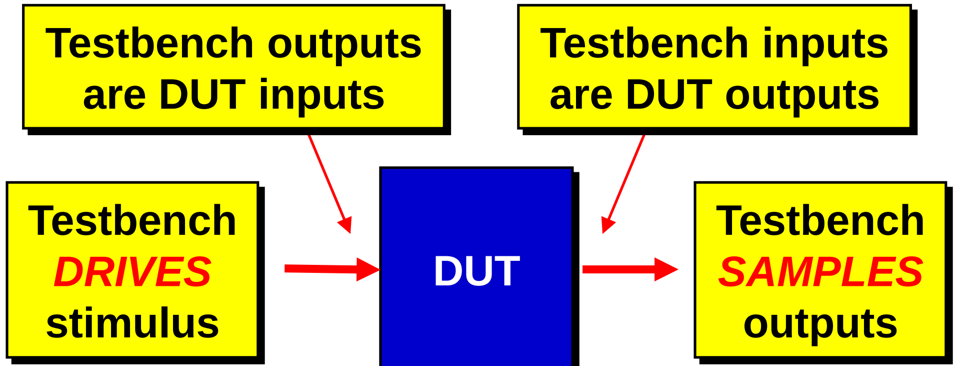
Definitions

- Used to specify synchronous sample & drive times
  - Input skew is for sampling
  - Output skew is for driving

Default input skew is `#1step` ← This is perfect!

Default output skew is `0` ← *This is bad!* Change this to 20% of `clk` cycle time





Testbench outputs are DUT inputs
Testbench inputs are DUT outputs
Testbench DRIVES stimulus
DUT
Testbench SAMPLES outputs

A `clocking` block encapsulates *when testbench* `inputs` *are sampled and when DUT stimulus is driven*

Example DUT - Program Counter

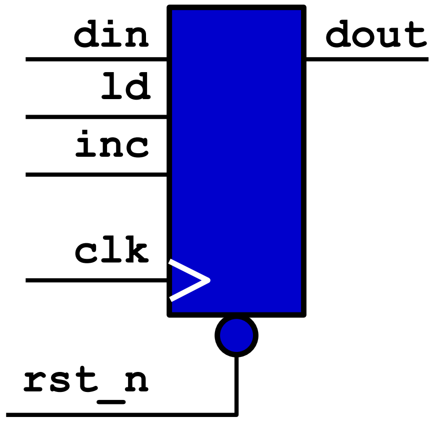
pcnt.sv

```
module pcnt (
  output logic [15:0] dout,
  input        [15:0] din,
  input               ld, inc, clk, rst_n);

  always_ff @(posedge clk, negedge rst_n)
    if    (!rst_n) dout <= '0;
    else if  (ld) dout <= din;
    else if (inc) dout <= dout + 1;
endmodule
```

Asynchronous control signal `rst_n`

Synchronous signals: `din`, `ld`, `inc`



Clocking Block In dut_if

CYCLE.sv

```
`ifndef CYCLE
  `define CYCLE 10
`endif
`ifndef Tdrive
  `define Tdrive #(0.2*`CYCLE)
`endif
`timescale 1ns/1ns
```

Define the Tdrive (drive time) to be 20% of the clock cycle

```
`include "CYCLE.sv"
interface dut_if (input clk);

  logic [15:0] dout;
  logic [15:0] din;
  logic        ld, inc, rst_n;

  clocking cb1 @(posedge clk);
    default input #1step output `Tdrive;
    input  dout;

    output din;
    output ld, inc, rst_n;
  endclocking

endinterface
```

The interface has a clocking block to control drive and sample timing

Inputs and outputs are declared in the dut_if and in the clocking block

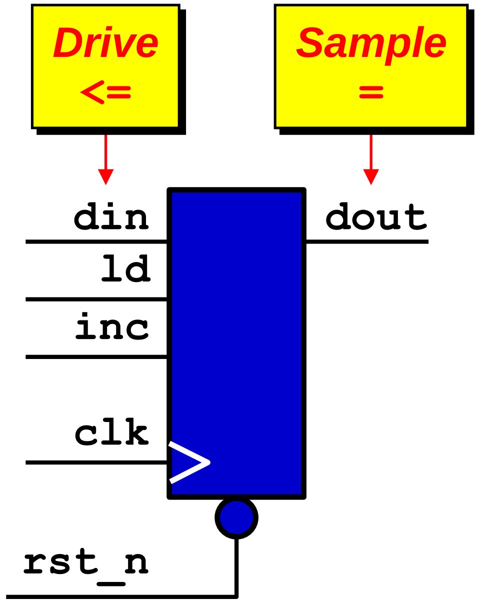
Data type, size and signal names

Signal names & direction

Sample DUT outputs #1step before the next posedge clk

Drive DUT inputs #2ns after the next posedge clk

20% of the clock cycle



Driving Stimulus

tb_driver.sv

Full example in the paper

Clocking block drives

```
  ...

task initialize();
  `uvm_info("RESET", "Initial reset", UVM_MEDIUM)
  vif.rst_n <= '0;
  vif.ld    <= '1;
  vif.inc   <= '1;
  vif.din   <= '1;
endtask


task drive_tr (trans1 tr);
  @(vif.cb1);
  `uvm_info("drive_tr", tr.convert2string(), UVM_HIGH)
  vif.cb1.din   <= tr.din;
  vif.cb1.ld    <= tr.ld;
  vif.cb1.inc   <= tr.inc;
  vif.cb1.rst_n <= tr.rst_n;
endtask

endclass
```

```
task run_phase(uvm_phase phase);
  trans1 tr;
  initialize();
  forever begin
    seq_item_port.get_next_item(tr);
    drive_tr(tr);
    seq_item_port.item_done();
  end
endtask
```

The `initialize` task does not use a `clocking` block (only called at time-0)

Synchronize to `@(posedge clk)` (the sample signal used by `cb1`)

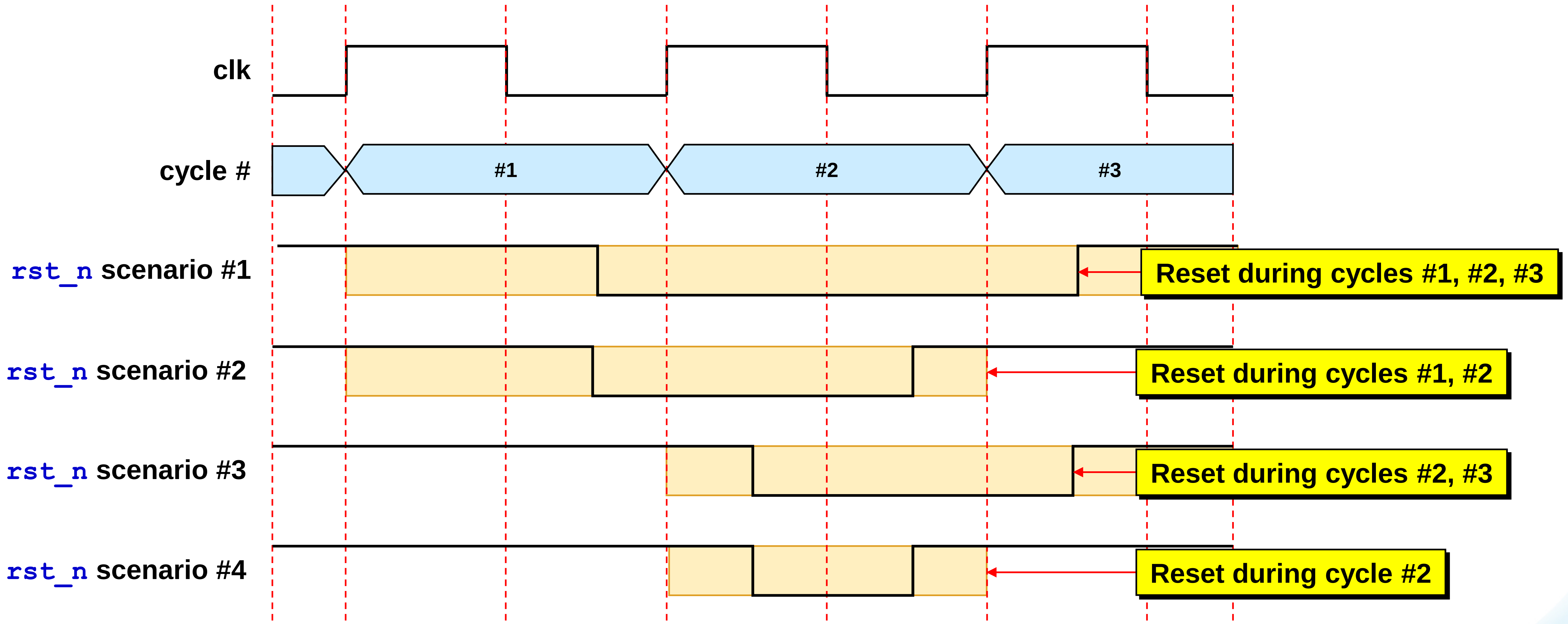
The `drive_tr` task DOES use a `clocking` block to control stimulus timing

Clocking drives: `vif.cb1.signal <=` ...

(drive) **<=**

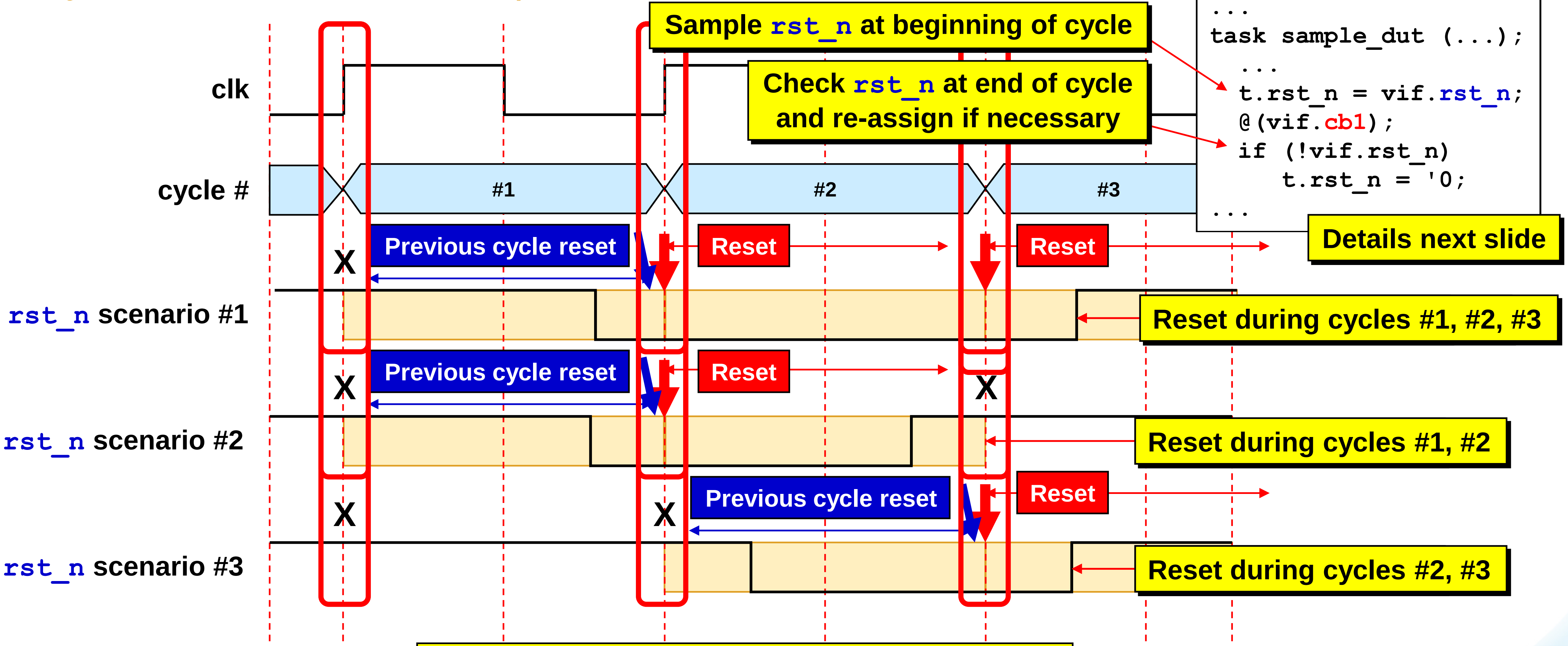
Asynchronous Control Signals

Asynchronous Reset Example



Asynchronous Control Signals

Asynchronous Reset Example - Scenarios #1 #2 #3



Sample rst_n at beginning of cycle
Check rst_n at end of cycle and re-assign if necessary
clk
cycle #
#1
#2
#3
Previous cycle reset
Reset
Reset
X
rst_n scenario #1
Reset during cycles #1, #2, #3
Previous cycle reset
Reset
X
X
rst_n scenario #2
Reset during cycles #1, #2
Previous cycle reset
Reset
X
X
rst_n scenario #3
Reset during cycles #2, #3
Details next slide

```
...
task sample_dut (...);
  ...
  t.rst_n = vif.rst_n;
  @(vif.cb1);
  if (!vif.rst_n)
     t.rst_n = '0;
...
```

Asynchronous control signals must be checked at the beginning and end of each cycle

Sampling Outputs & Inputs

tb_monitor.sv

Full example in the paper

Clocking block samples

```
  ...
  task run_phase(uvm_phase phase);
    trans1 tr;
    tr = trans1::type_id::create("tr");
    forever begin
      sample_dut(tr);
      aport.write(tr);
    end
  endtask

  task sample_dut (output trans1 tr);
    trans1 t;
    t = trans1::type_id::create("t");
    t.din   = vif.din;
    t.ld    = vif.ld;
    t.inc   = vif.inc;
    t.rst_n = vif.rst_n;
    @(vif.cb1);
    if (!vif.rst_n) t.rst_n = '0;
    t.dout  = vif.cb1.dout;
    tr      = t;
    `uvm_info("sample_dut", tr.convert2string(), UVM_FULL)
  endtask
endclass
```

(drive) <=

Clocking drives: `vif.cb1.signal <= ...`
Clocking sample: `... = vif.cb1.signal`

(sample) =

`sample_dut` is assumed to be sync-ed to the `posedge clk`

Inputs used to predict next output

DUT inputs are sampled on the `posedge clk`

DOES NOT use `clocking` block timing

Sync to `posedge clk`

Re-test async control inputs

Sample DUT outputs `#1step` before `posedge clk`

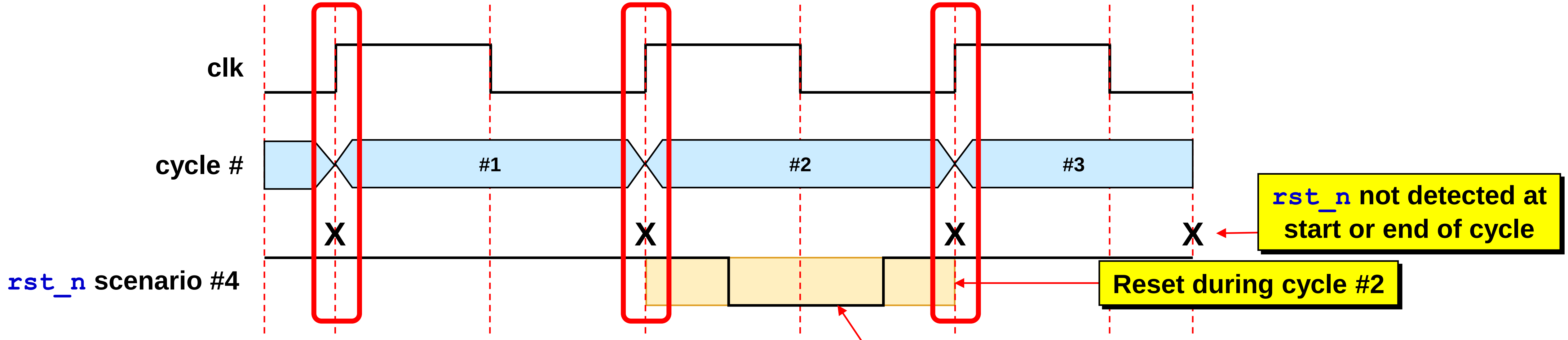
Sampling `dout` *DOES* use `clocking` block timing *(and =)*

Actual outputs used to compare against predicted (expected) outputs

Assign `t` to task output `tr`

Asynchronous Control Signals

Sub-Cycle Asynchronous Reset



rst_n not detected at start or end of cycle

Reset during cycle #2

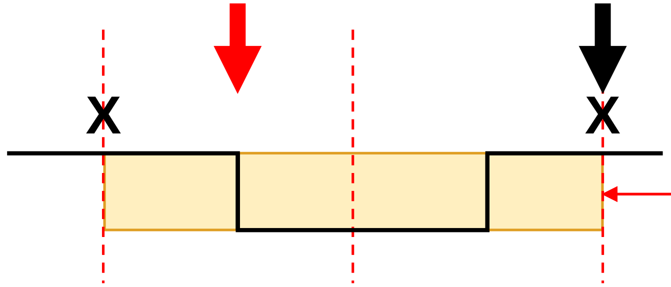
"*Sticky bit*" detection required in `interface`

Next slide

Asynchronous Control Signals

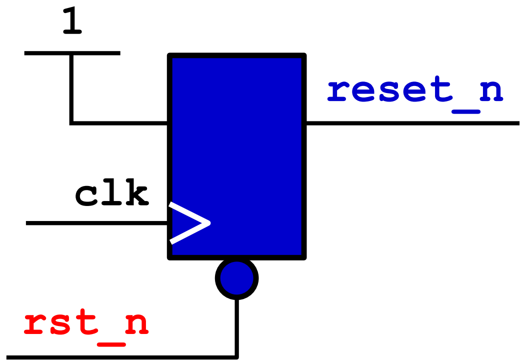
Sub-Cycle Asynchronous Reset

rst_n scenario #4



Reset mid-cycle

interface "*Sticky bit*" behavioral model



```
interface dut_if (input clk);
  logic [15:0] dout;
  logic [15:0] din;
  logic        ld, inc, rst_n;
  logic        reset_n;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) reset_n <= '0;
    else        reset_n <= '1;

  clocking cb1 @(posedge clk);
    default input #1step output `Tdrive;
    input  dout;
    output din;
    output ld, inc, rst_n;
    input  reset_n;
  endclocking
endinterface
```

"*Sticky bit*" detection required in interface

Sticky reset_n to capture short rst_n pulses

reset_n set to 1 on next posedge clk when rst_n is disabled

Sample reset_n in interface ...

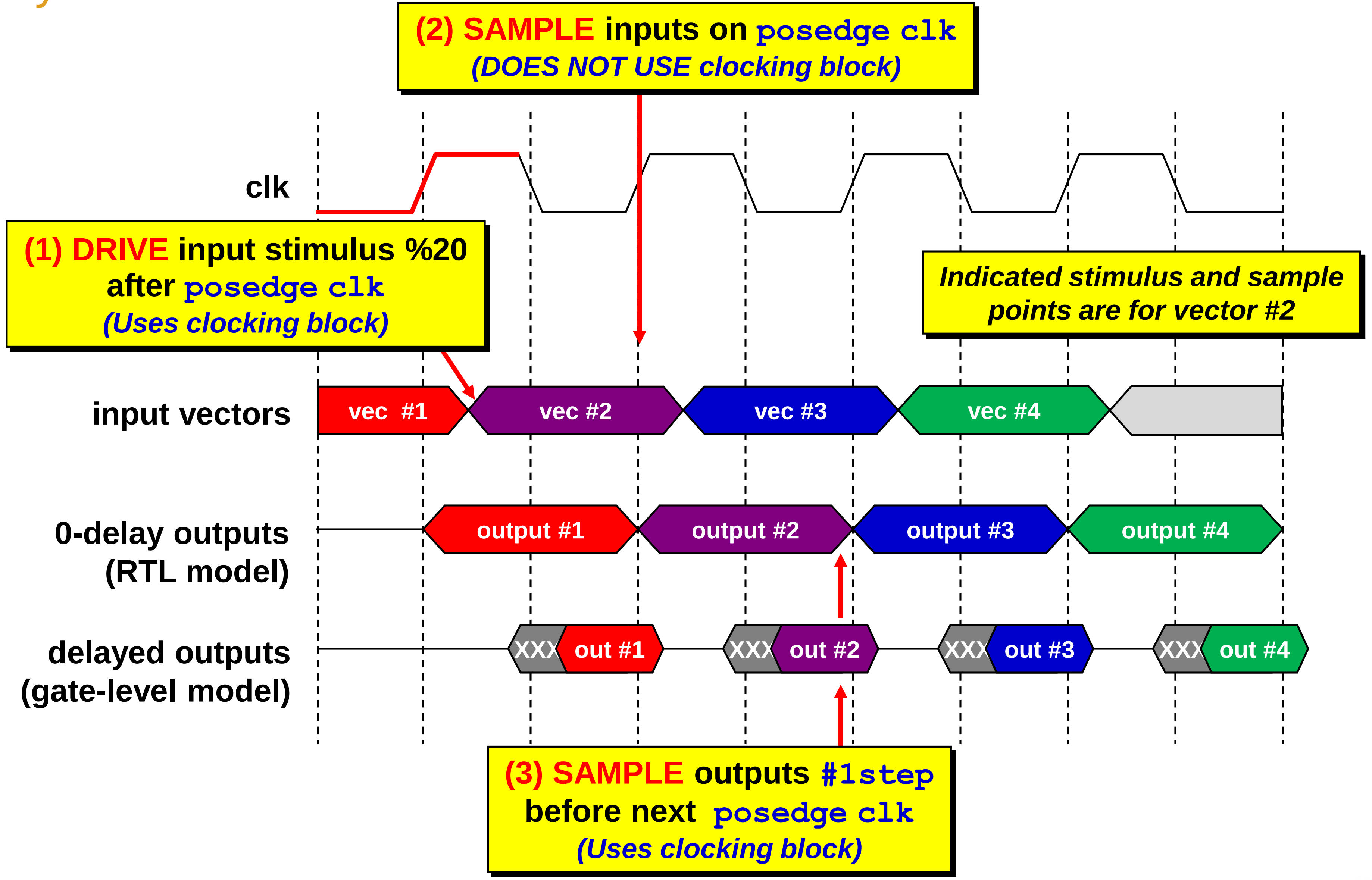
... test reset_n #1step before posedge clk to assign rst_n at end of cycle

tb_monitor modification

```
...
task sample_dut (...);
  ...
  t.rst_n = vif.rst_n;
  @(vif.cb1);
  if (!vif.cb1.reset_n)
    t.rst_n = '0;
...
```

Three Important Timing Points

Summary



(2) SAMPLE inputs on `posedge clk`
(DOES NOT USE clocking block)
clk
(1) DRIVE input stimulus %20 after `posedge clk`
(Uses clocking block)
Indicated stimulus and sample points are for vector #2
input vectors
vec #1
vec #2
vec #3
vec #4
0-delay outputs (RTL model)
output #1
output #2
output #3
output #4
delayed outputs (gate-level model)
XXX
out #1
XXX
out #2
XXX
out #3
XXX
out #4
(3) SAMPLE outputs `#1step` before next `posedge clk`
(Uses clocking block)

SystemVerilog Programs

A SystemVerilog-2005 "enhancement" that should die!

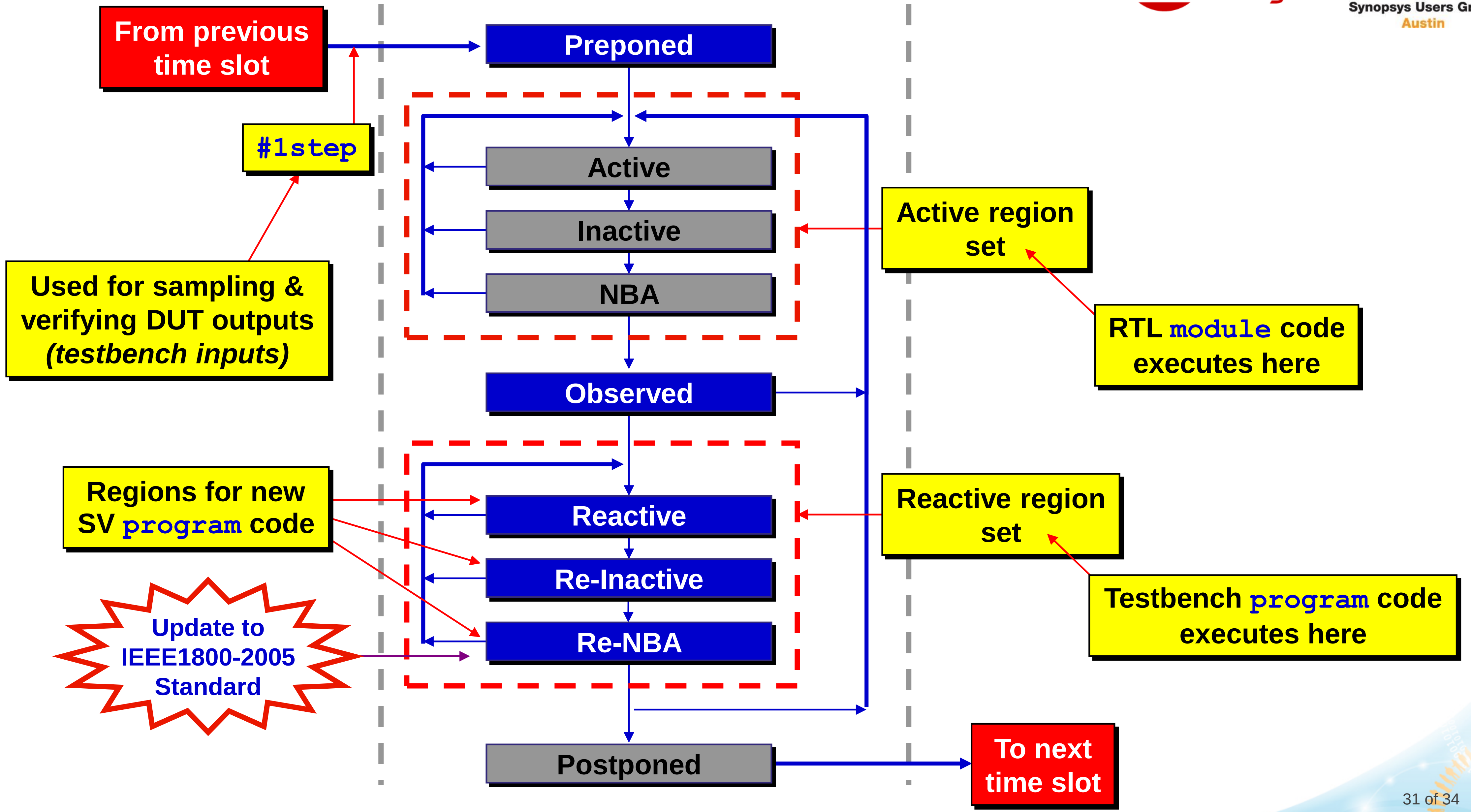
SystemVerilog Programs

Added to SystemVerilog-2005

- The **`program`** solves a problem that never should have been solved!
- The **`program`** has multiple restrictions when interacting with a **`module`**
- The **`program`** adds confusion to SystemVerilog event scheduling
- The **`program`** is not needed
- The **`program`** *should just die!*

The `program` is not used by UVM and is not needed!

SystemVerilog Event Scheduling


From previous time slot
Preponed
#1step
Active
Inactive
NBA
Active region set
RTL module code executes here
Used for sampling & verifying DUT outputs (testbench inputs)
Observed
Regions for new SV program code
Reactive
Re-Inactive
Re-NBA
Reactive region set
Testbench program code executes here
Update to IEEE1800-2005 Standard
Postponed
To next time slot

Summary & Conclusions

- Time-0 is a tricky place in SystemVerilog simulations ← **Use nonblocking assignments at time-0**
- There are three important timing values to be used by verification engineers
- (1) When to drive stimulus - use time-budgeting ← **20% after the active clock edge**
- (2) When to sample inputs
  - On active clock edge for synchronous inputs
  - On active clock edge and on next active clock edge for asynchronous control signals
  - Might require ***sticky-bit*** technique for sub-cycle asynchronous control signal pulses
- (3) When to sample DUT outputs - `#1step` before next active clock edge

  Use `#1step` in a `clocking` block
- Use `clocking` blocks in an `interface` to help control testbench timing
- Never use (*or quit using*) the SystemVerilog `program`

  ***DEATH* to the SystemVerilog `program`!!**

Acknowledgements

Thanks!

- Jeff Montesano for his review and suggested improvements to the paper and presentation

- My colleague and friend, Jonathan Bromley for exchanges of ideas on clocking blocks and for his previous, co-authored SNUG-Austin paper detailing the behavior of clocking blocks and recommended guidelines

Synopsys Users Group
Austin

Thank You

Applying Stimulus & Sampling Outputs UVM Verification Testing Techniques

Clifford E. Cummings

Sunburst Design, Inc.

World-class SystemVerilog & UVM Verification Training

Life is too short for bad or boring training!

September 29, 2016

SNUG Austin

Free IEEE SystemVerilog-2012 LRM @
`http://standards.ieee.org/getieee/1800/download/1800-2012.pdf`