SystemVerilog Assertions - Bindfiles & Best Known Practices for Simple SVA Usage

Clifford E. Cummings
Sunburst Design, Inc.

World-class SystemVerilog & UVM Verification Training

Life is too short for bad or boring training!

10th EDITION DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Cliff Cummings recommended Expert Services Company

PARADIGM® WORKS

Good reference papers:

SystemVerilog Assertions – Bindfiles & Best Known Practices for Simple SVA Usage
`www.sunburst-design.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf`

SystemVerilog Assertions - Design Tricks and SVA Bind Files
`www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf`

accellera SYSTEMS INITIATIVE™

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Cliff Cummings is Vice President of Training at Paradigm Works and Founder of Sunburst Design. Cliff teaches world-class SystemVerilog, UVM,CDC and synthesis training classes.

Cliff has more than 40 years of ASIC, FPGA and system design experience and more than 30 years of combined Verilog, SystemVerilog, UVM verification, synthesis, and methodology training experience.

Cliff has taught expert Verilog, SystemVerilog, Design Synthesis, CDC and UVM Verification to thousands of engineers world-wide and has presented more than 60 papers on these topics.

Cliff holds a BSEE from BYU and an MSEE from Oregon State University.

SystemVerilog Assertions - Bindfiles & Best Known Practices for Simple SVA Usage

Clifford E. Cummings
Sunburst Design, Inc.

World-class SystemVerilog & UVM Verification Training

Life is too short for bad or boring training!

10th EDITION DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Cliff Cummings recommended Expert Services Company

PARADIGM® WORKS

Good reference papers:

SystemVerilog Assertions – Bindfiles & Best Known Practices for Simple SVA Usage
`www.sunburst-design.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf`

SystemVerilog Assertions - Design Tricks and SVA Bind Files
`www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf`

accellera SYSTEMS INITIATIVE™

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Agenda

- Motivation for the 2016 SVA paper
- The SVA *"Circle of Life!"*
- How much training should be taken?
- Who should add assertions?
- Bindfiles
- Long labels for debugging
- Simple assertion macros
- Concurrent -vs- immediate assertions
- `|-> ##1` -vs- `|=>`
- Adoption issues
- Summary - 10 Guidelines

Motivation and who should add assertions

Bindfiles and why you should use them

Techniques to code assertions faster with fewer errors

Much more in the papers!

Motivation

Why the paper?

- Harry Foster is the co-inventor of OVL

Open Verification Library
Assertions created from Verilog modules

These were the first HDL concurrent assertions

- Harry is co-author of the first Assertion Based Design Book

Co-authors: Adam Krolnik & David Lacey

- Harry & Cliff - SVA Verification Seminars in U.S. and Europe in 2010
- Harry & Cliff exchanged assertion ideas - Best Known Practices
- 2010 Seminar attendees shared feedback and their best practices

The 2016 paper shares new Best Known Practices

The Ups & Downs of SVA

- The SVA "circle of life"
  - Engineers get excited about SVA capabilities
  - Engineers take SVA training
  - Engineers start to use SVA
  - Engineers find SVA to be too verbose
  - Engineers *abandon* SVA

"The Disillusioned Design Engineer cycle!!"

aargh

Engineers need to learn simple, non-verbose techniques

SVA Fundamentals

Guidelines

- Start learning and using SVA after 2-3 hours of lecture and 1-3 hours of labs
- Use ***bindfiles*** to add assertions to a design
- Use long, descriptive labels to:
  - document the assertions
  - accelerate debugging using waveform displays
- Use simple macros to:
  - efficiently add concise assertions
  - reduce assertion coding efforts
  - reduce assertion syntax errors
- Use concurrent assertions but avoid immediate assertions
- Use `|-> ##1` implications instead of `|=>` implications

How Much SVA Training?

- 2-day SystemVerilog Assertion (SVA) training is too much

You can be productive after just 2-3 hours of training

- Inefficient SVA coding styles are shown in books and training
- Most engineers should take 2 hours of SVA training with labs

Learn the simple techniques

Reduce SVA coding efforts

Learn best-practice tricks

Learn styles that avoid mistakes

- **Guideline #1:** Start learning and using SVA after 2-3 hours of lecture and 1-3 hours of labs.

Full-time assertion engineers *SHOULD* take multi-day training

Who should add assertions?

- Some sources suggest:
  - Designers should add immediate assertions
  - Verification engineers should add concurrent assertions

Cliff strongly *disagrees* !!

- Harry Foster from 2010 seminars:
  - Designers should add low-level assertions
  - Verification engineers should add high-level assertions
  - Harry included other "*Assertion Stakeholders*" responsibilities

Cliff strongly *AGREES* !!

Later **- Cliff recommends:**
- use concurrent assertions
- avoid immediate assertions

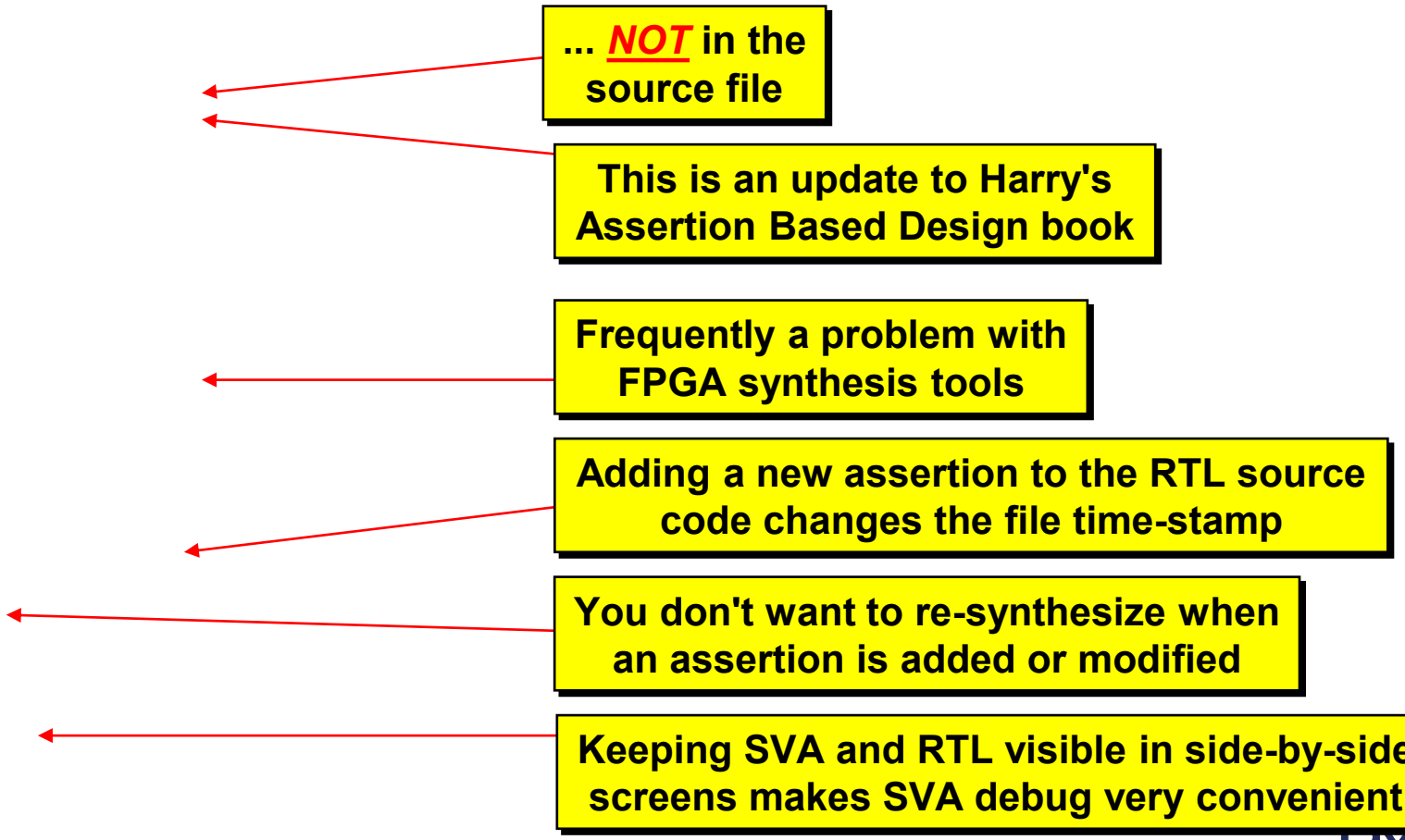
SVA Bindfiles

Place Assertions Into Bindfiles

- **Guideline #2**: bindfiles - use them!
- **Guideline #3**: Inline SVA code - avoid it!

- Place Assertions into a bindfile

- Reasons:
  - Some EDA tools do not recognize assertion code
  - Many designs have a `Makefile` to execute synthesis runs
  - Visibility of SVA in bindfiles



FIFO DUT code

Includes subtle bugs

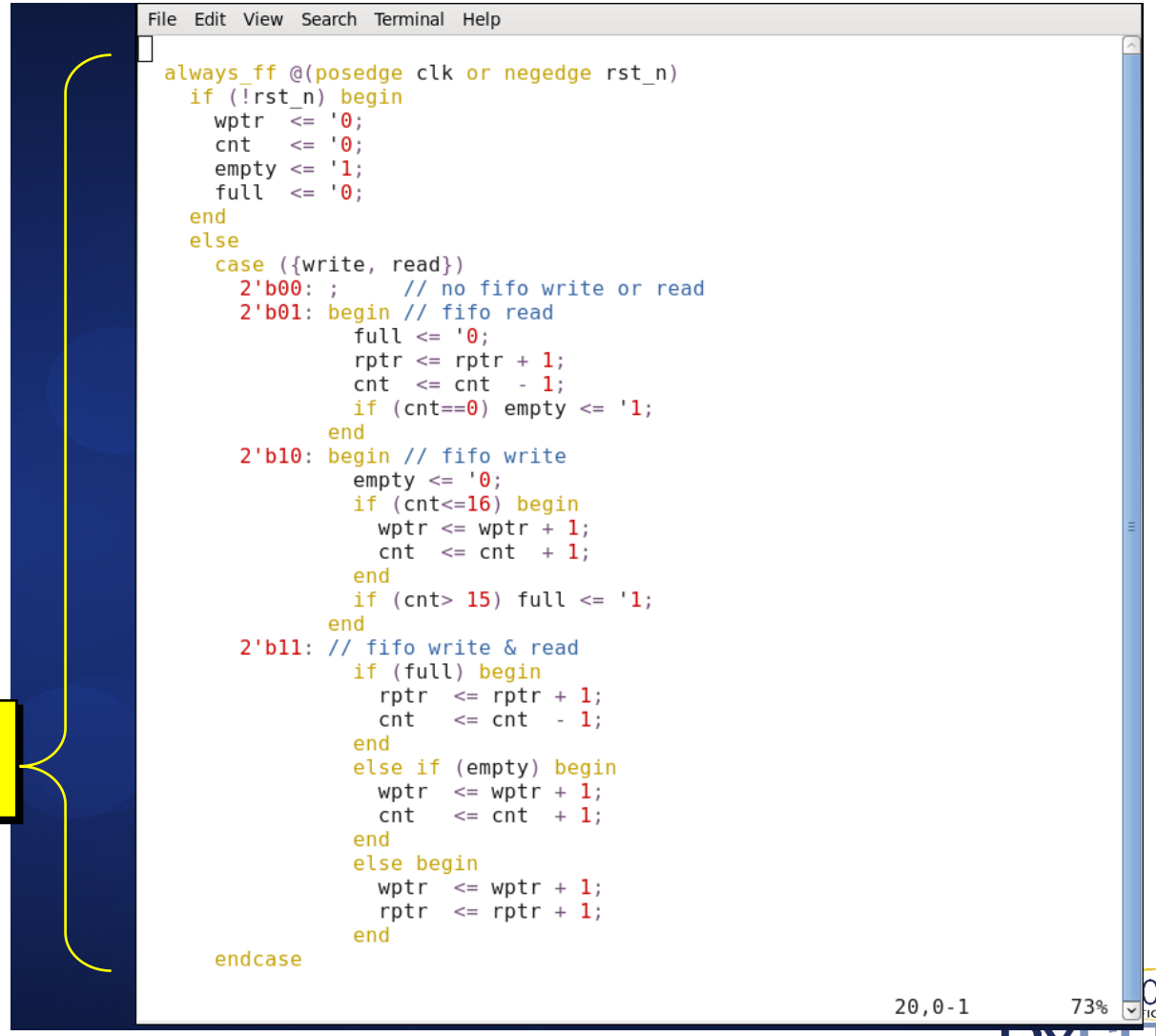
- Screen shot of FIFO
- Some sources suggest:
  - Place assertions next to corresponding DUT code
  - Better for documentation

Easier said than done!

My experience differs

Cannot insert concurrent assertions into the middle of this block

Very large `always_ff` block of code



FIFO DUT code

Scrolling back-and-forth

- Placing assertions next to applicable DUT code
  - May not be easy
  - May require code-scrolling to view
  - ... and more code -scrolling!
  - Harder to view RTL code in one screen

One solution: place assertions in separate `bind` file

Open both assertions file and RTL file side-by-side

Actual screenshot

File Edit View Search Terminal Help

```
logic [3:0] wptr, rptr;
logic [3:0] cnt;

ERR_FIFO_WORD_COUNTER_IS_NEGATIVE:
  `assert_clk_xrst (not (cnt<0));

ERR_FIFO_SHOULD_BE_EMPTY:
  `assert_clk (cnt==0 |-> empty);

ERR_FIFO_SHOULD_NOT_BE_EMPTY:
  `assert_clk_xrst (cnt>0 |-> !empty);

ERR_FIFO_DID_NOT_GO_EMPTY:
  `assert_clk_xrst (cnt==1 && read && !write |-> ##1 empty);

ERR_FIFO_EMPTY_READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
  `assert_clk_xrst (empty && read && !write |-> ##1 empty);

ERR_FIFO_EMPTY_READ_CAUSED_RPTR_TO_CHANGE:
  `assert_clk_xrst (empty && read && !write |-> ##1 $stable(rptr));

ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
  `assert_clk(!rst_n |->
    (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin
    wptr  <= '0;
    cnt   <= '0;
    empty <= '1;
    full  <= '0;
  end
  else
    case ({write, read})
      2'b00: ;     // no fifo write or read
      2'b01: begin // fifo read
               full <= '0;
               rptr <= rptr + 1;
               cnt  <= cnt  - 1;
               if (cnt==0) empty <= '1;
             end
      2'b10: begin // fifo write
               empty <= '0;
               if (cnt<=16) begin
```

Visibility of SVA in bindfiles

Modern Laptops have Wide Screens

Actual screenshot

Or view assertions file on 2nd screen

Both files visible on same laptop screen

Easier to scroll RTL code while viewing assertions

Personal experience: **This reduces SVA coding and debug time**

```
File  Edit  View  Search  Terminal  Help
  ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
    `assert_clk(!rst_n |->
      (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));

  ERR_FIFO_SHOULD_BE_FULL:
    `assert_clk_xrst (cnt>15 |-> full);

  ERR_FIFO_SHOULD_NOT_BE_FULL:
    `assert_clk_xrst (cnt<16 |-> !full);

  ERR_FIFO_SHOULD_BE_EMPTY:
    `assert_clk_xrst (cnt==0 |-> empty);

  ERR_FIFO_SHOULD_NOT_BE_EMPTY:
    `assert_clk_xrst (cnt>0 |-> !empty);

  ERR_FIFO_DID_NOT_GO_FULL:
    `assert_clk_xrst (cnt==15 && write && !read |-> ##1 full);

  ERR_FIFO_DID_NOT_GO_EMPTY:
    `assert_clk_xrst (cnt==1 && read && !write |-> ##1 empty);

  ERR_FIFO_FULL_WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
    `assert_clk_xrst (full && write && !read |-> ##1 full);

  ERR_FIFO_FULL_WRITE_CAUSED_WPTR_TO_CHANGE:
    `assert_clk_xrst (full && write && !read |-> ##1 $stable(wptr));

  ERR_FIFO_EMPTY_READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
    `assert_clk_xrst (empty && read && !write |-> ##1 empty);

  ERR_FIFO_EMPTY_READ_CAUSED_RPTR_TO_CHANGE:
    `assert_clk_xrst (empty && read && !write |-> ##1 $stable(rptr));

  ERR_FIFO_WORD_COUNTER_IS_NEGATIVE:
    `assert_clk_xrst (not (cnt<0));

  ERR_FIFO_READWRITE_ILLEGAL_FIFO_FULL_OR_EMPTY:
    `assert_clk_xrst (write && read |-> ##1 !full && !empty);
endmodule
                                                    59,3          Bot
```

```
File  Edit  View  Search  Terminal  Help

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin
    wptr  <= '0;
    cnt   <= '0;
    empty <= '1;
    full  <= '0;
  end
  else
    case ({write, read})
      2'b00: ;      // no fifo write or read
      2'b01: begin // fifo read
               full <= '0;
               rptr <= rptr + 1;
               cnt  <= cnt  - 1;
               if (cnt==0) empty <= '1;
             end
      2'b10: begin // fifo write
               empty <= '0;
               if (cnt<=16) begin
                 wptr <= wptr + 1;
                 cnt  <= cnt  + 1;
               end
               if (cnt> 15) full <= '1;
             end
      2'b11: // fifo write & read
               if (full) begin
                 rptr  <= rptr + 1;
                 cnt   <= cnt  - 1;
               end
               else if (empty) begin
                 wptr  <= wptr + 1;
                 cnt   <= cnt  + 1;
               end
               else begin
                 wptr  <= wptr + 1;
                 rptr  <= rptr + 1;
               end
    endcase
                                                    20,0-1        73%
```

SystemVerilog Assertion Bind Files

Coding Techniques

Assertions Module

Create the Assertions Wrapper

```
`define assert_clk_xrst(arg) \  ...
`define assert_clk(arg) \  ...

module fifo1_asserts(
  input [7:0] dout, din,
  input [4:0] cnt,
  input [3:0] wptr, rptr,
  input       empty, read, full,
  input       write, clk, rst_n);

  ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
    `assert_clk (!rst_n |->   ...

  ERR_FIFO_SHOULD_BE_FULL:
    `assert_clk (cnt>15 |-> full);
  ...
endmodule
```

Move the FIFO assertions to a separate module

In general, all assertion module ports will be `inputs`

`module` - *`ports`* - `endmodule` used to "wrap" the assertions

Exact same assertion code as would be used directly in the design

Instantiated Assertions Module Using .*

```
module tb1;
  ...
  fifo1 u1    (.*);

  ...
endmodule
```

`fifo1` instantiated into `tb1`

```
module fifo1 (
  output logic [7:0] dout,
  output logic       full, empty,
  input  logic       write, read, clk, rst_n,
  input  logic [7:0] din);
  logic [7:0] fifomem [0:15];
  logic [3:0] wptr, rptr;
  logic [4:0] cnt;
  ...

  fifo1_asserts p1 (.*) ;
  ...
endmodule
```

Aaargh!! Slide-typo !!

```
module fifo1_asserts(
  input [7:0] dout, din,
  input [4:0] cnt,
  input [3:0] wptr, rptr,
  input       empty, read, full,
  input       write, clk, rst_n);
  ...
endmodule
```

FIFO assertions module

`fifo1_asserts` instantiated into `fifo1`

All `fifo1_asserts` port names match `fifo1` signal names so `.*` instantiation is possible

Do not modify RTL model - `fifo1_asserts` instantiation *not allowed !!*

Assertions Module Binding #1

1st Style -Testbench bind

How does the bind command work??
(next slide)

```
module tb1;
  ...
  fifo1 u1    (.*);
  bind fifo1  fifo1_asserts p1 (.*);
  ...
endmodule
```

1st Style - in testbench

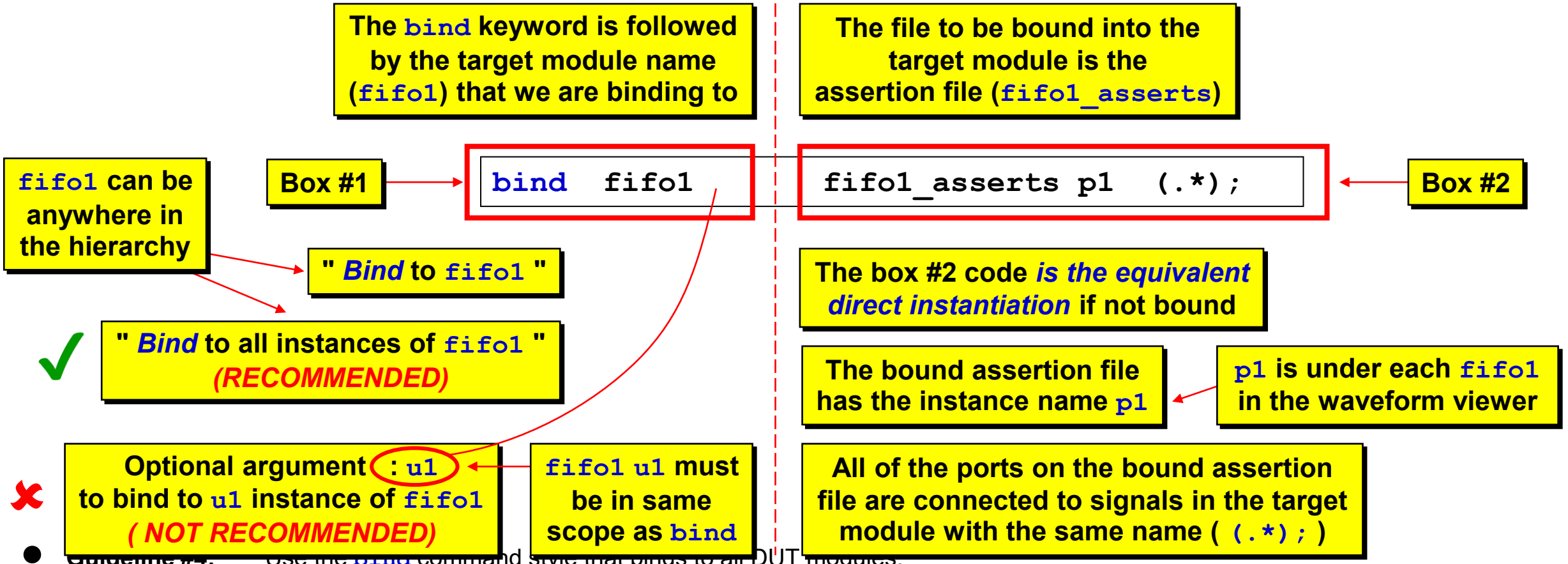
(2) Use `bind` in the testbench to indirectly instantiate the assertions into the RTL model

```
module fifo1 (
  output logic [7:0] dout,
  output logic       full, empty,
  input  logic       write, read, clk, rst_n,
  input  logic [7:0] din);
  logic [7:0] fifomem [0:15];
  logic [3:0] wptr, rptr;
  logic [4:0] cnt;
  ...

  fifo1_asserts p1 (.*);
  ...
endmodule
```

NOTE: these two code fragments are identical

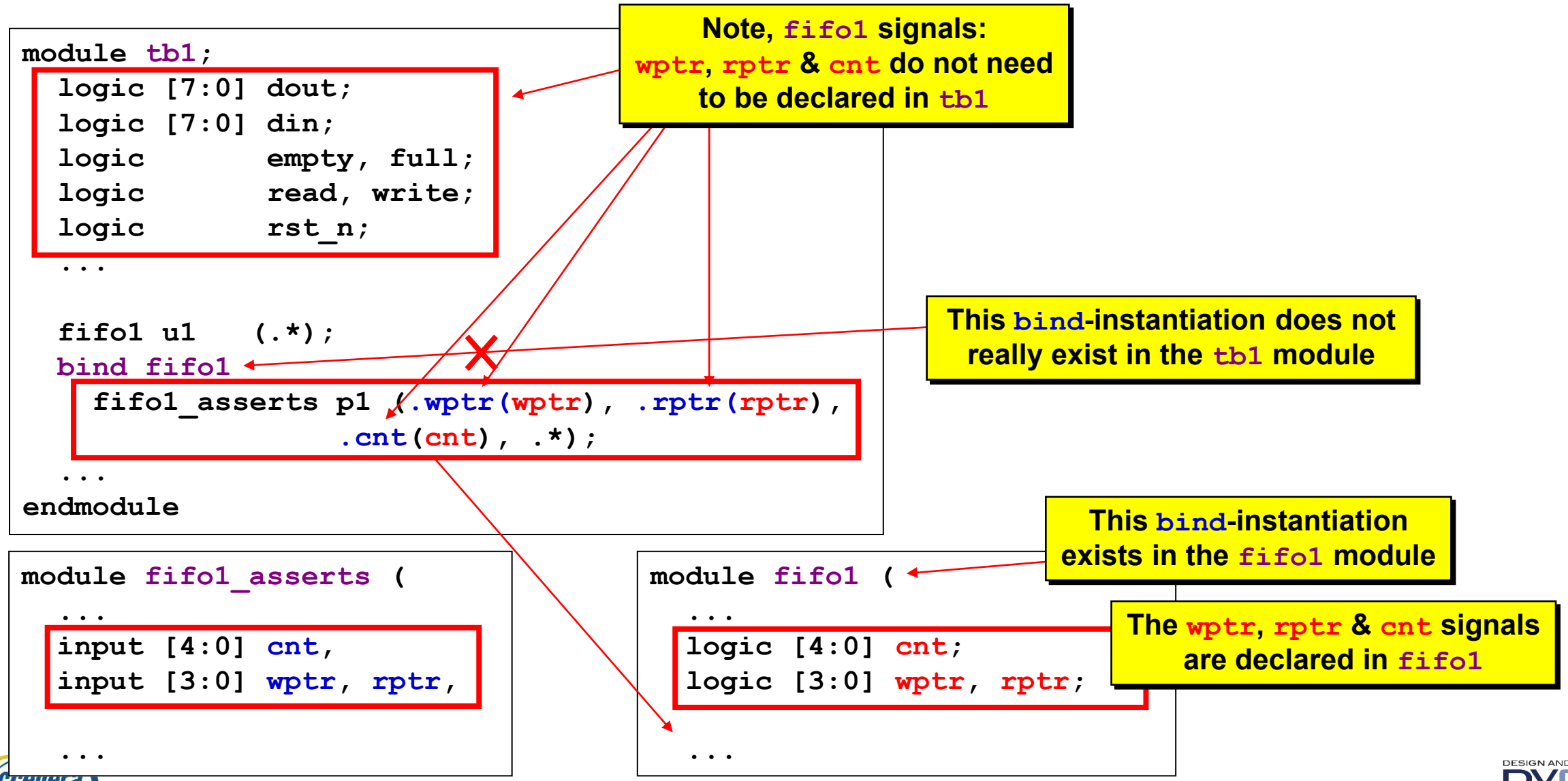
(1) Remove the instantiation from the RTL model

Bind Command - Closer Look



- **Guideline #4:** Use the `bind` command style that binds to all DUT modules, not the `bind` style that only binds to specified instances

Bindfile Signal Declarations?



Assertions Module Binding #2

2nd Style - Separate bindfile

```
module tb1;
  ...
  fifo1 u1 (.*);
  ...
endmodule
```

```
module fifo1 (
  (all port declarations) );
  ...
```

```
module bindfiles;
  bind fifo1 fifo1_assert p1 (.*);
endmodule
```

```
module fifo1_asserts (
  (all input declarations) );
  ...
```

fifo1 assertions

When simulating with assertions:
Two top-level modules:
tb1 and bindfiles

Put the bind command in a separate bindfiles.sv file

To simulate tb1 and fifo1:

```
irun +sv -f run1.f
```

```
vcs -sverilog -R -f run1.f
```

```
vlog -sv -f run1.f
vsim tb1
```

run1.f

```
tb1.sv
fifo1.sv
```

Pure RTL -
No assertion code

Call both Top Level Modules when running Questa vsim

To simulate tb1, fifo1, bindfiles and fifo1_asserts:

```
vlog -sv +acc -f run1a.f
vsim tb1 bindfiles -assertdebug
```

run1a.f

```
-f run1.f
bindfiles.sv
fifo1_assert.sv
```

RTL & assertions

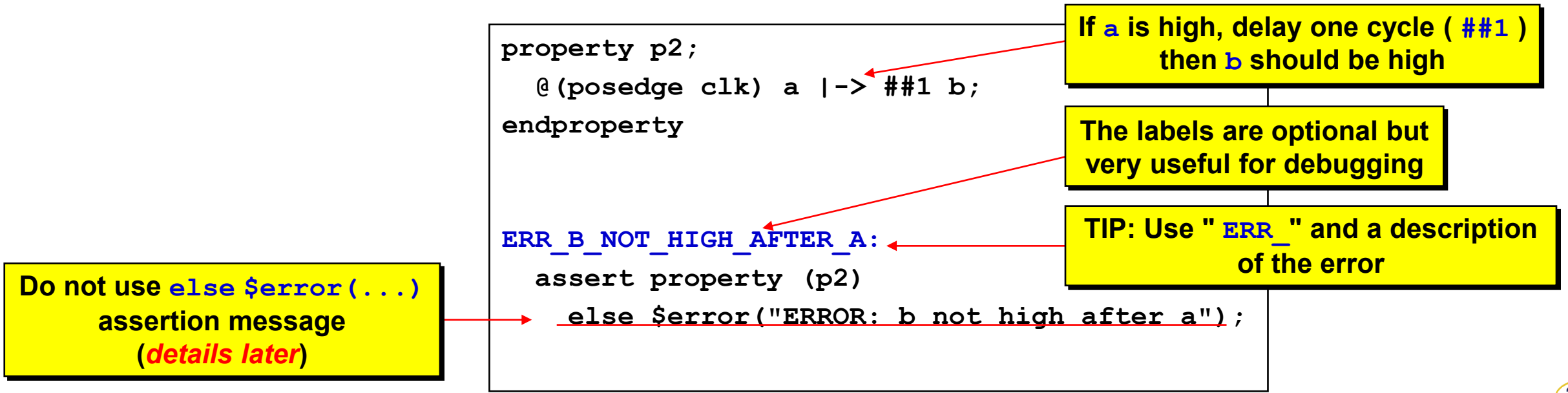
Assertion Labeling Technique

Use For Rapid Waveform Debugging

Assert Labels -vs- Messages

- **Guideline #2:** Add descriptive labels to your assertion code
- **Guideline #3**: Use label names that start with "`ERR`" or "`ERROR`" and then include a short sentence to describe the failure

- Long descriptive labels help debug `assert` failures

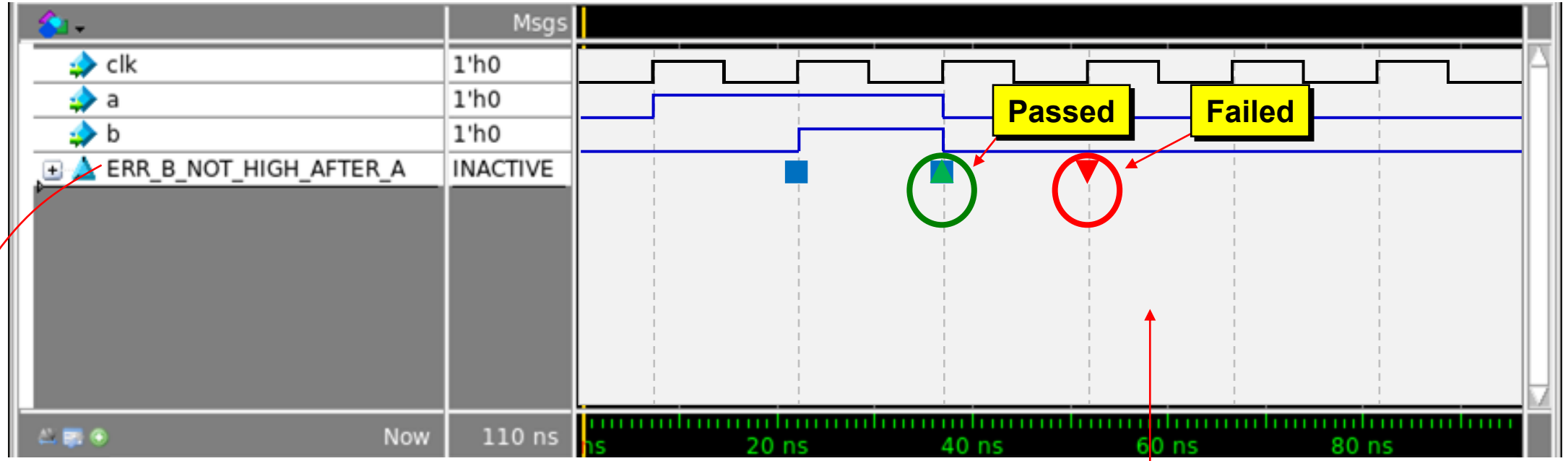


Assertion Labels & Messages

Simple Example & Waveform Display

- **Guideline #7**: In general, do not use `$error(...)` or `$display(...)` messages in assertion action blocks

Use descriptive labels instead



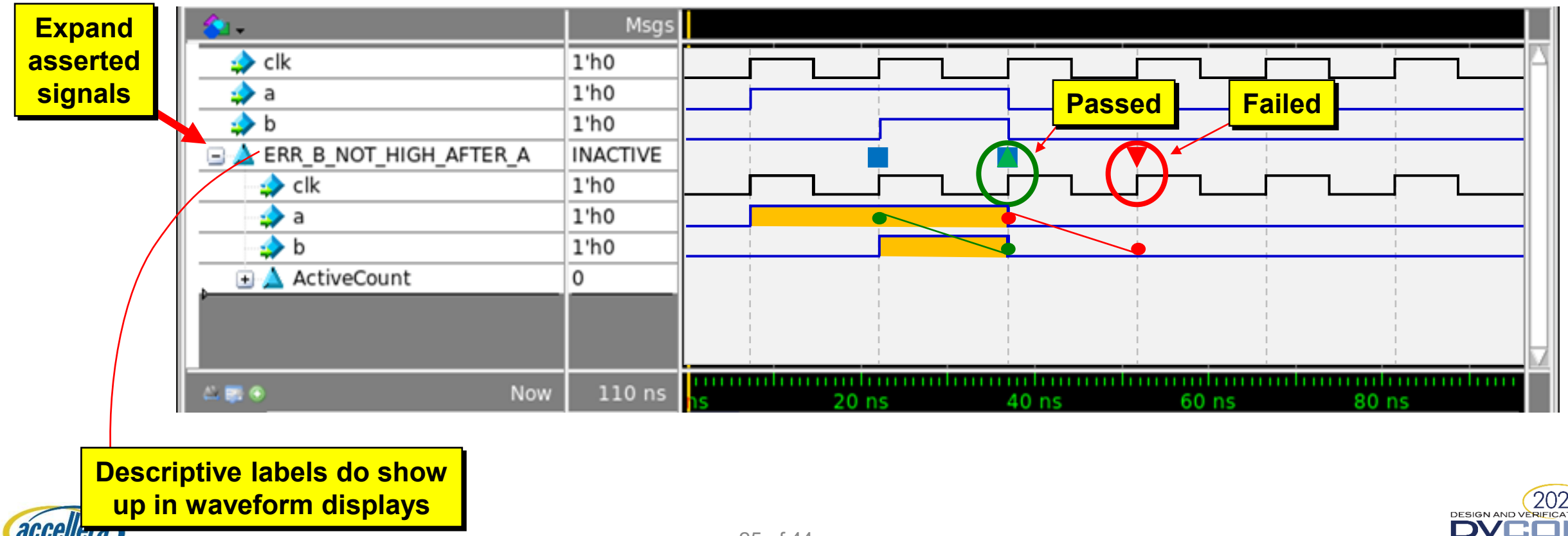
Descriptive labels do show up in waveform displays

Error messages do not show up in waveform displays

Assertion Labels & Messages

Simple Example & Waveform Display

- **Guideline #7**: In general, do not use `$error(...)` or `$display(...)` messages in assertion action blocks

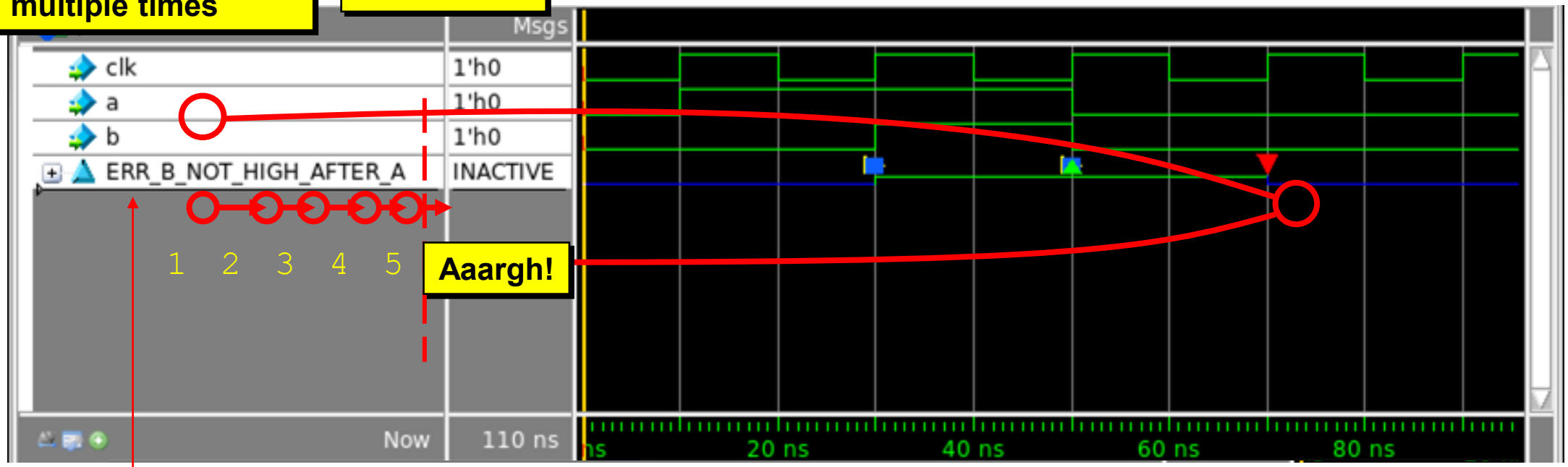


Assertion Labels in Waveforms

Beginner -vs- Expert Debugging

Beginners:
Grow the waves window multiple times

Very painful to watch !!



Full labels are *not visible* when waves are first displayed

How do Experts do it?

Experts:
(1) Grow the waves window once
(2) Find the end of the labels
(3) Shrinks the window to the labels

Just two window adjustments and ready to debug!

Assertion Macros

To Reduce Assertion Coding Effort

To Reduce Assertion Coding Mistakes

Good reference papers:

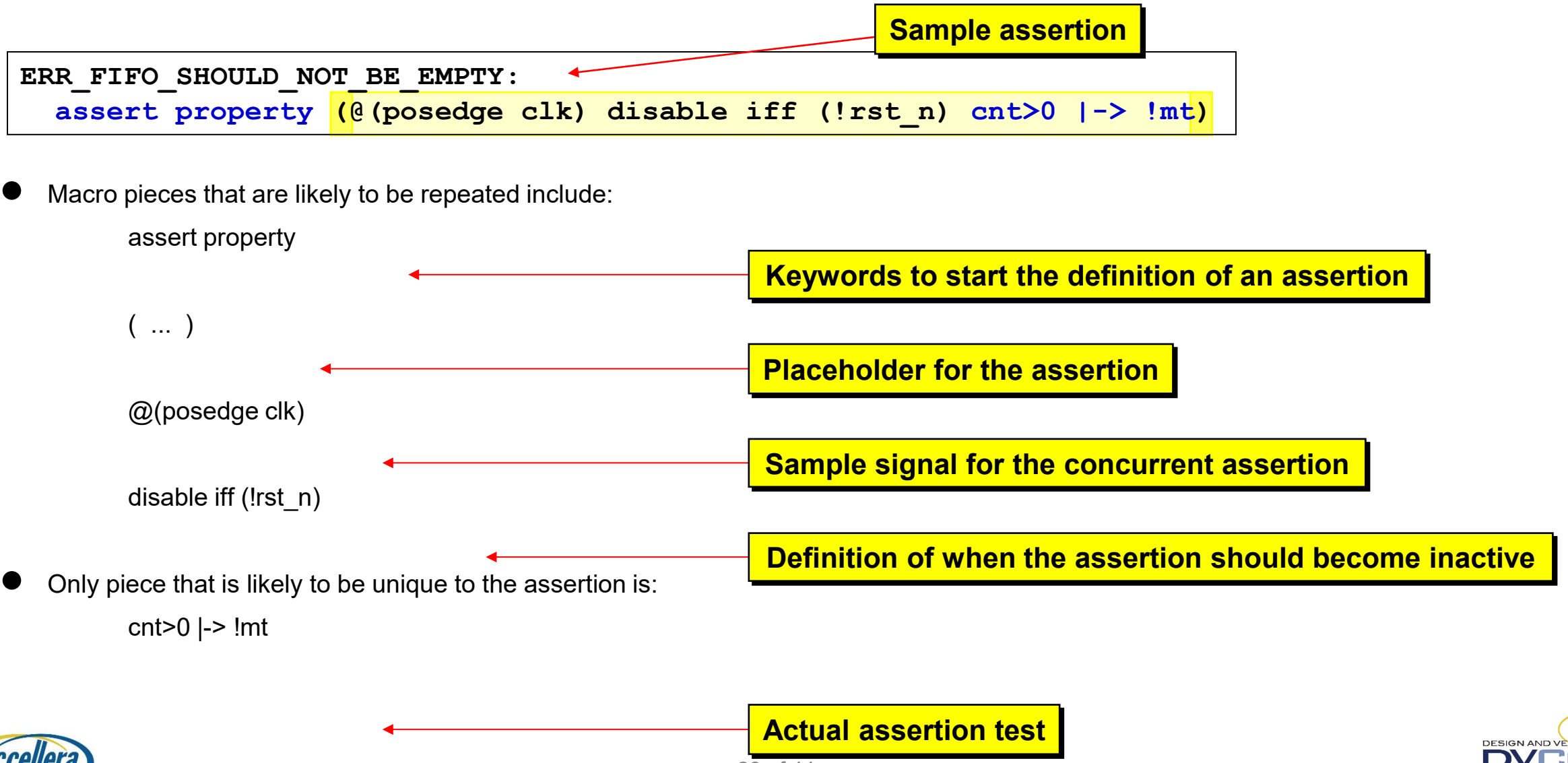
SystemVerilog Assertions – Bindfiles & Best Known Practices for Simple SVA Usage

`www.sunburst-design.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf`

SystemVerilog Assertions - Design Tricks and SVA Bind Files

`www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf`

Example Assertion Pieces



- Macro pieces that are likely to be repeated include:

  assert property

  (...)

  @(posedge clk)

  disable iff (!rst_n)

- Only piece that is likely to be unique to the assertion is:

  cnt>0 |-> !mt

Assertion Macro Style

Useful Macro Definitions

- **Guideline #8**: Use macros to reduce SVA coding efforts and errors

```
`ifndef ASSERT_MACROS

`define ASSERT_MACROS

`define assert_clk(arg) \
    assert property (@(posedge clk) arg)

`define assert_clk_xrst(arg) \
    assert property (@(posedge clk) disable iff (!rst_n) arg)

`endif
```

`include assert_macros.sv

ASSERT_MACROS are defined if not already defined

assert_clk macro definition

; is not part of the macro definitions

When macros are used, add the end-of-expression ;

assert_clk_xrst macro definition

Assertion Coding Styles

You Choose!

```
property p1;
  @(posedge clk)
  disable iff (!rst_n)
  cnt>0 |-> !mt;
endproperty

ERR_FIFO_SHOULD_NOT_BE_EMPTY:
  assert property (p1);
```

Define a separate property and then `assert` it

Common assertion coding style #1

```
ERR_FIFO_SHOULD_NOT_BE_EMPTY:
  assert property (@(posedge clk) disable iff (!rst_n) cnt>0 |-> !mt)
```

`assert` a property without a separate definition

Common assertion coding style #2

Improved, concise, assertion coding style

```
`define assert_clk_xrst( arg ) \
  assert property (@(posedge clk) disable iff (!rst_n) arg )
```

Define an `assert_clk_xrst` macro ...

```
ERR_FIFO_SHOULD_NOT_BE_EMPTY:
  `assert_clk_xrst(cnt>0 |-> !mt);
```

... then use the `` `assert_clk_xrst `` macro multiple times *(very concise!)*

Concurrent -vs- Immediate Assertions

Concurrent -vs-Immediate Assertions

- **Guideline #9**: Use concurrent assertions - avoid immediate assertions

- Recommended usage:
  - **Concurrent assertions** for all *synchronous* activity

Recommended: put concurrent assertions in bindfiles

Check signals once per clock cycle at the end of the cycle

Allows signals to settle to final values

In general - you don't care if the signals are glitching between clock edges

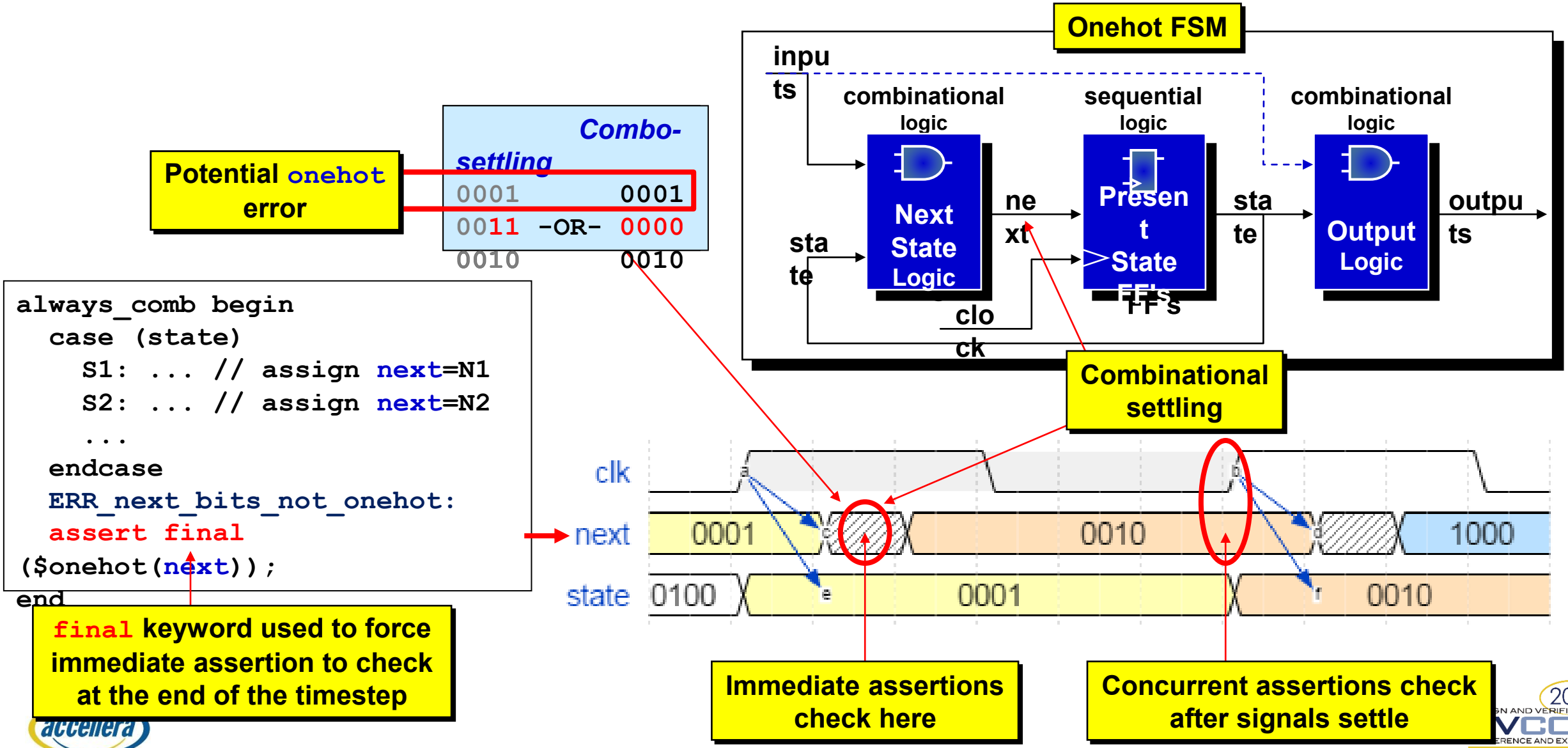
The testbench does not check signals mid-cycle!

  - **Immediate assertions** to test *asynchronous control* signals

Recommended: also put many immediate assertions in bindfiles !!

Concurrent -vs-Immediate Assertions

Don't test during combinational settling



Concurrent -vs-Immediate Assertions

Problematic Example

```
always_ff @(posedge clock or negedge rst_n)
begin
  assert (!$isunknown(rst_n))
    else $error("unknown value on rst_n");
  if (!rst_n) q <= 0;
  else        q <= d;
end
```

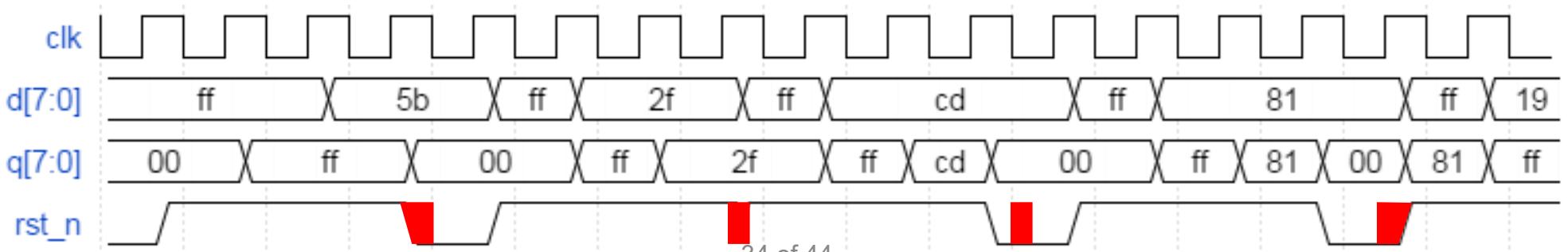
Commonly shown immediate assertion

Assert the `rst_n` control line is never an `X` or `Z`

Trap asynchronous reset-`X` errors

Two detected assertion errors

With 100 flip-flops, message-to-error ratio is very bad



Concurrent -vs-Immediate Assertions

Problematic Example

Commonly shown immediate assertion

```
always_ff @(posedge clk or negedge rst_n) begin
  assert (!$isunknown(rst_n))
    else
      $error("unknown value on rst_n");
  ...
```

With 100 flip-flops, message-to-error ratio is very bad

400 lines of error messages *(for 2 errors)*

```
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[12] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[11] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[10] File: register.sv Line: 14
                st_n
                est.r[9] File: register.sv Line: 14
                st_n
                est.r[8] File: register.sv Line: 14
                st_n
                est.r[7] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[6] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[5] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[4] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[3] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[2] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[1] File: register.sv Line: 14
** Error: unknown value on rst_n
   Time: 69 ns  Scope: reg_test.r[0] File: register.sv Line: 14
```

Concurrent -vs-Immediate Assertions

Added Label & Removed $error

```
always_ff @(posedge clock or negedge rst_n)
begin
  ERR_reset_went_unknown:
  assert (!$isunknown(rst_n));
  if (!rst_n) q <= 0;
  else        q <= d;
end
```

1st improvement: remove $error message and add label

```
** Error: Assertion error.
   Time: 39 ns  Scope: reg_test.r[99].ERR_reset_went_unknown File: register.sv Line: 12
```

Label included in error message

2 lines of error messages with label

```
  assert (!$isunknown(rst_n))
    else $error("unknown value on rst_n");
```

... but with 100 flip-flops

Still very bad

$error message

No label

```
** Error: unknown value on rst_n
   Time: 39 ns  Scope: reg_test.r[99] File: register.sv Line: 14
```

2 lines of error messages with no label

Concurrent -vs-Immediate Assertions

Immediate Assertion Moved to bindfile

```
module DUT_asserts (
  // input declarations, including reset
);

  always @(negedge rst_n) begin
    ERR_reset_went_unknown:
    assert(!$isunknown(rst_n))
  end
...
endmodule
```

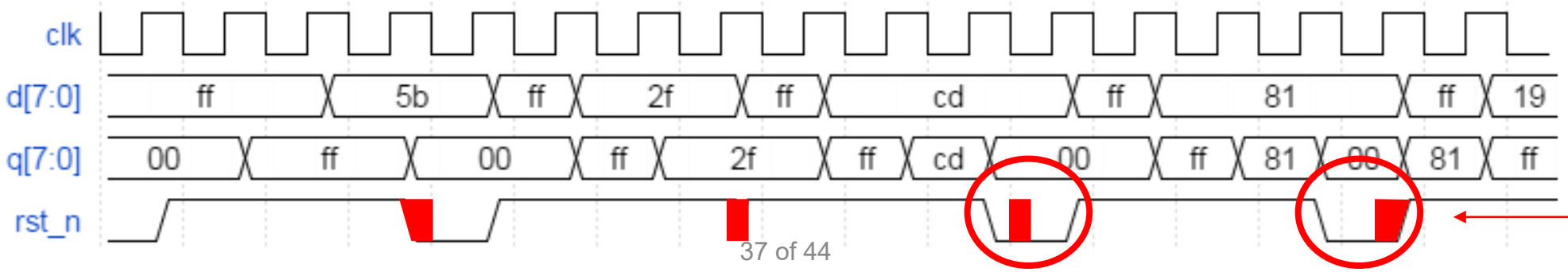
2nd improvement: move to *bindfile*

Two detected assertion errors

4 lines of error messages *(for 2 errors)*

Much better!

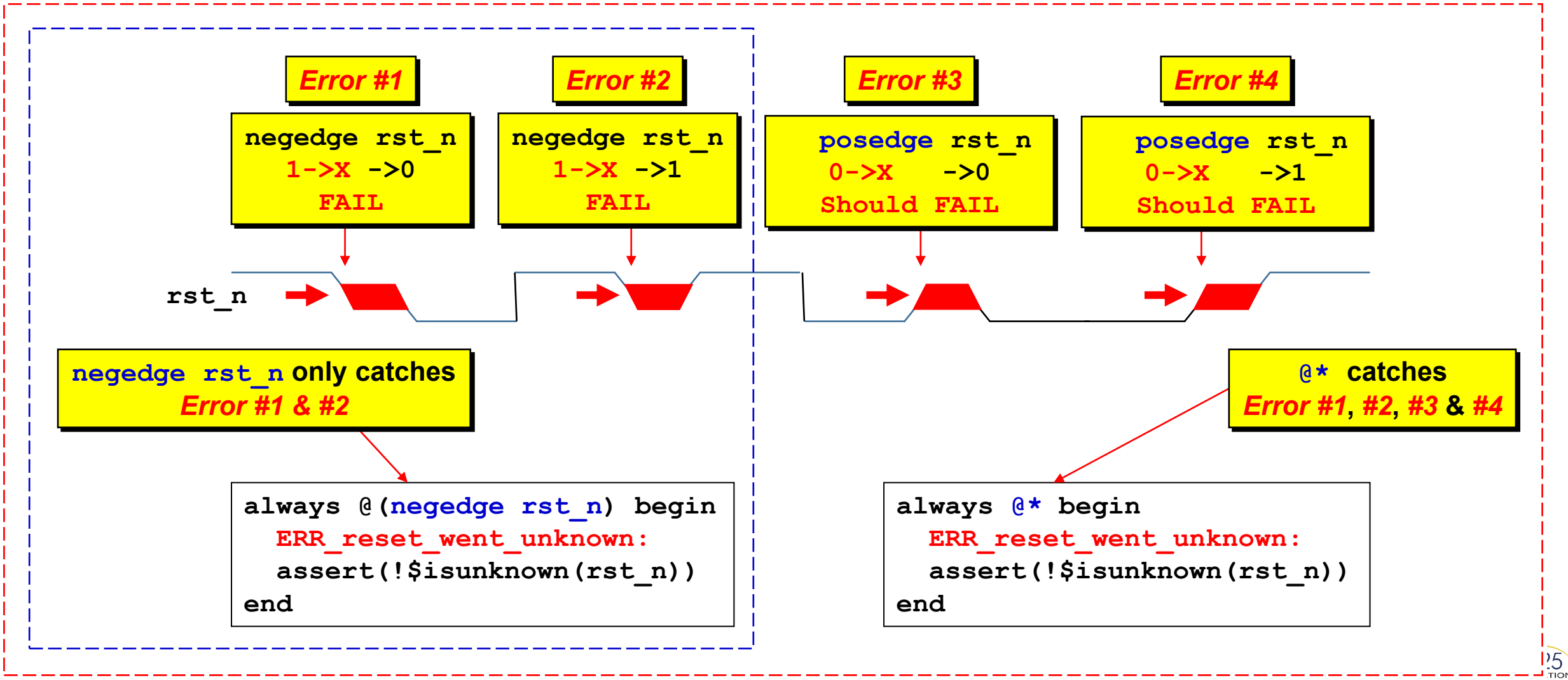
Still only capturing half of the potential errors



Missed!

Reset-X Possibilities

Four Possible Error Scenarios



Concurrent -vs-Immediate Assertions

Test both reset edges in bindfile

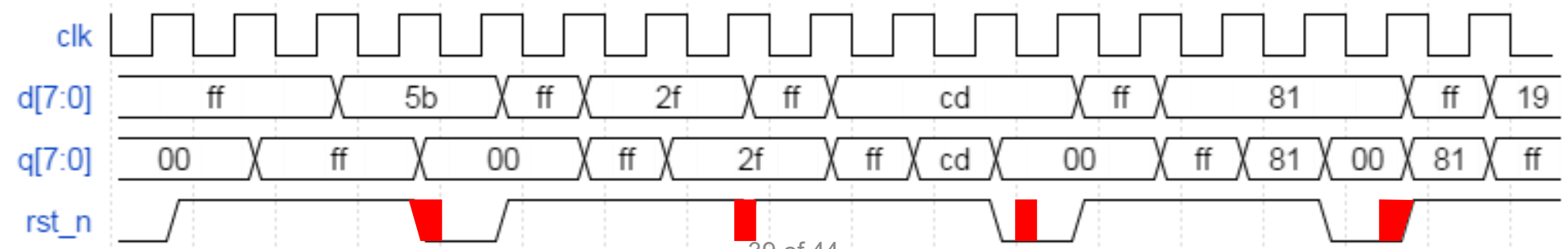
```
module DUT_asserts (
  // input declarations, including reset
);

  always @* begin
    ERR_reset_went_unknown:
    assert(!$isunknown(rst_n))
  end
...
endmodule
```

3rd improvement: capture on *BOTH* edges of `rst_n`

Now captures *ALL* of the potential errors

Four assertion errors

8 lines of error messages *(for 4 errors)*



Concurrent -vs-Immediate Assertions

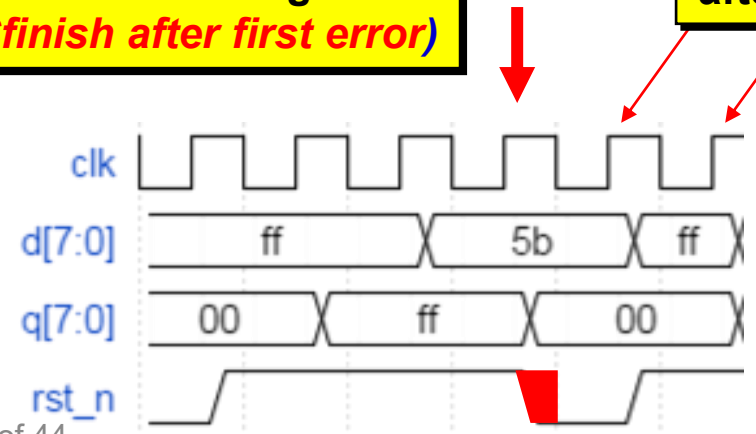
Test both reset edges in bindfile and Stop

```
module DUT_asserts (
  // input declarations, including reset
);

  always @* begin
    ERR_reset_went_unknown:
    assert(!$isunknown(rst_n))
      else begin
        repeat(2) @(posedge clk);
        $finish;
      end
  end
...
endmodule
```

Now assertion terminates 2 cycles after first error

2 lines of error messages (*$finish after first error*)

Terminates 2 cycles after assertion failure



Overlapping Implication Operator

Superset of Non-Overlapping Implication Operator

- **Guideline #10**: Use `|-> ##1` implications and not `|=>` implications

Allan Hunter of ARM shared this technique at DAC

Equivalent Implications

`|->`	*(nothing)*
`|-> ##1`	`|=>`
`|-> ##2`	`|=> ##1`
`|-> ##3`	`|=> ##2`
`|-> ##4`	`|=> ##3`
...	...

IF a is high ... ... **`b` should be high in the next cycle**

```
property p3;
  @(posedge clk) a |=> b;
endproperty
```

IF a is high ... ... **wait one cycle and then `b` should be high in *that* cycle**

```
property p4;
  @(posedge clk) a |-> ##1 b;
endproperty
```

Can reduce confusion

Does reduce assertion coding errors

Mixing Up |=> and |->

Easy to Confuse

Large text: Easy to see |=> |-> difference

```
ERR_FIFO_SHOULD_BE_FULL:
`assert_clk (cnt>15 |-> full);

ERR_FIFO_SHOULD_NOT_BE_FULL:
`assert_clk (cnt<16 |-> !full);

ERR_FIFO_DID_NOT_GO_FULL:
`assert_clk_xrst (cnt==15 && write && !read |=> full);
```

Smaller text: *Not-bad* to see |=> |-> difference

```
ERR_FIFO_SHOULD_BE_FULL:
`assert_clk (cnt>15 |-> full);

ERR_FIFO_SHOULD_NOT_BE_FULL:
`assert_clk (cnt<16 |-> !full);

ERR_FIFO_DID_NOT_GO_FULL:
`assert_clk_xrst (cnt==15 && write && !read |=> full);
```

Micro text: Good luck!!

```
ERR_FIFO_SHOULD_BE_FULL:
`assert_clk (cnt>15 |-> full);

ERR_FIFO_SHOULD_NOT_BE_FULL:
`assert_clk (cnt<16 |-> !full);

ERR_FIFO_DID_NOT_GO_FULL:
`assert_clk_xrst (cnt==15 && write && !read |=> full);
```

To avoid errors, just use |->

Engineers Don't Like Change!

- Conventional wisdom has been:
  - Place assertions in the RTL code
  - Add assertion `$error` messages
  - Use `|=>` implications

- Change is hard!

- If you do choose to make these c*I know!*

I knew these methods for years before I implemented *all* of them

  - They will reduce assertion coding errors and usage frustration
  - They will reduce assertion coding *efforts*
  - They will help debug the RTL quicker
  - They will motivate designers to add more assertions

You Decide!
Choose the *Best Known Practices* that will benefit your team

Conclusions

Summary of Guidelines (1 of 2)

- **Guideline #1**: Start learning and using SVA after 2-3 hours of lecture and 1-3 hours of labs
- **Guideline #2**: ***bindfiles*** - use them!
- **Guideline #3**: Inline SVA code - avoid it!
- **Guideline #4:** Use the `bind` command style that binds to all DUT modules, not the `bind` style that only binds to specified instances
- **Guideline #5:** Add descriptive labels to your assertion code

Conclusions

Summary of Guidelines (2 of 2)

- **Guideline #6**: Use label names that start with "`ERR`" or "`ERROR`" and then include a short sentence to describe the failure
- **Guideline #7**: In general, do not use `$error(...)` or `$display(...)` messages in assertion action blocks
  - Use long, descriptive labels to (a) document the assertions, and (b) accelerate debugging using waveform displays
- **Guideline #8**: Use macros to reduce SVA coding efforts and errors
- **Guideline #9**: Use concurrent assertions - avoid immediate assertions
- **Guideline #10**: Use `|-> ##1` implications and not `|=>` implications

SystemVerilog Assertions - Bindfiles & Best Known Practices for Simple SVA Usage

Clifford E. Cummings
Sunburst Design, Inc.

World-class SystemVerilog & UVM Verification Training

Life is too short for bad or boring training!

10th EDITION DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Cliff Cummings recommended Expert Services Company

PARADIGM® WORKS

Good reference papers:

SystemVerilog Assertions – Bindfiles & Best Known Practices for Simple SVA Usage
`www.sunburst-design.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf`

SystemVerilog Assertions - Design Tricks and SVA Bind Files
`www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf`

accellera SYSTEMS INITIATIVE™

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA