If Chained Implications in Properties Weren't So Hard, They'd be Easy

Don MIlls

Microchip Technology Inc.

Chandler, AZ, USA

don.mills@microchip.com

mills@lcdm-eng.com

www.microchip.com

2009

USER2USER

Introductions:
Who is Don Mills anyway

Don Mills

MICROCHIP

- Over 20 Years in the Industry
- Over 30 ASICS/Designs
- Consultant/Trainer for 10 years with
  - Sutherland HDL
  - Sunburst Design
- Experience with the “big” three simulators
- Member of the IEEE 1800 SystemVerilog BC and EC committees
- Member of the IEEE 1801 UPF committee
- Presented numerous papers at various conference
  - Go to www.lcdm-eng.com to access papers
- Co-author of “Verilog and SystemVerilog Gotchas”

Don Mills

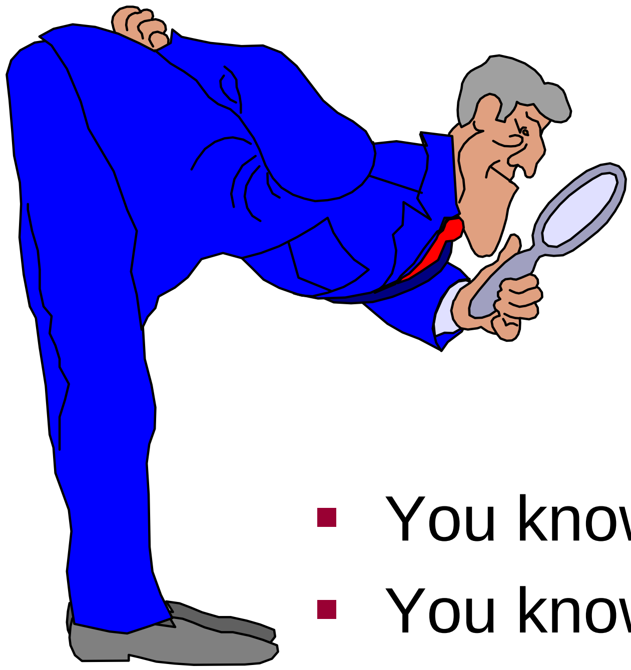
Mentor and SV Stuff

- Questa Advanced Support for SV
  - SystemVerilog for Design
  - Verification – the best in the industry (in my opinion)
    - Constrained Random / OOP based test environment
    - Verification Environments
  - Coverage
    - Reports, tables, charts
    - UCDB – driving the standard
  - Assertions
    - ATV – Assertion Thread Viewer

One of MENTORS Best Kept Secret!!!

Don Mills

Assumptions



- You know basically what assertions are
- You know basically what properties are
- You know basically what sequences are
- You know what range repetition is

What is an Implication? Quick review

Don Mills

MICROCHIP

- Implication operators – in property expressions only
- Provide a conditional test for a sequence
  - If the condition is true, the sequence is evaluated
  - If the condition is false, the sequence is not evaluated
- **|->** overlapped implication: sequence evaluation starts immediately
- **|=>** non-overlapped implication: sequence evaluation starts at the next clock

```
property bus_req_prop6;
       @(posedge clk) req |-> ##[1:5] grant;
endproperty:bus_req_prop6
```

If `req` is false, I don't care about `grant`

Only test for `grant` if `req` tests true

Implication Terminology

- Antecedent – the condition or expression before the implication operator
- Consequent – the expression following the implication operator
- Vacuous success – Name for the don’t-care condition when the antecedent is false

```
property example_5;
  @(posedge clk) antecedent_sequence_expression |->
                           consequent_property_expression;
endproperty:example_5
```

Don Mills

Why use chained implications?

- Chained implications
  - Allow for multi-level (or hierarchical) conditioning
    - Like nested if-then

```
Pseudo code example:
if chip_en then
  if bank_en then
    if mem_en then
      verify mem
```

Don't care about mem unless all conditions are true

```
property p_chain;
  @(posedge clk) chip_en |->  bank_en |->  mem_en |-> mem;
endproperty:p_chain;
```

  - Can share local variables between the different levels

Don Mills

The Spec and the Code

- Monitor number of cycles between a start and an end point
  - Verify the number of cycles between the start and the end points is less than the max allowed

```
`define TRUE 1

property p_max_cycles;
  int v_cnt;
  @(posedge clk) ($rose(start), v_cnt = 0) |->
          (`TRUE, v_cnt++)[*0:$] ##1 done |->
                               (v_cnt <= MAX);
endproperty:p_max_cycles

ap_max_cycles: assert property (p_max_cycles);
```

If only this property worked …

Simpler Version of the Code

```
property p_chain;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain;

ap_chain: assert property (p_chain);
```

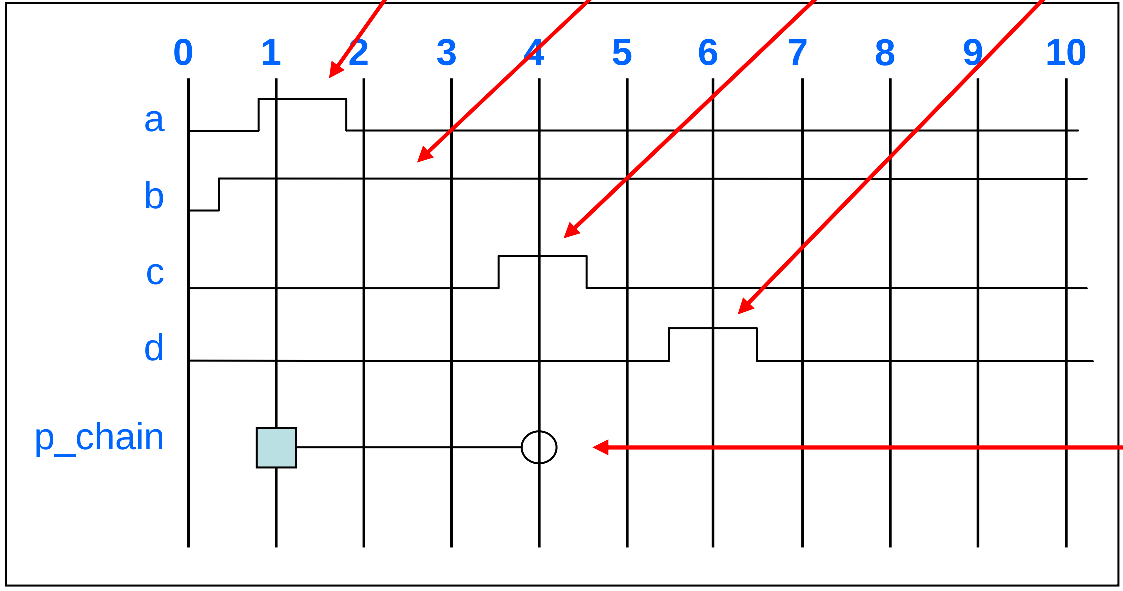
- With a chained implication
  - What is the antecedent?
  - What is the consequent?

$$\underbrace{\texttt{\$rose(a)}}_{\texttt{A1}}\ \texttt{|->}\ \underbrace{\underbrace{\texttt{b[*0:\$] \#\#1 c}}_{\texttt{a2}}\ \texttt{|->}\ \underbrace{\texttt{d;}}_{\texttt{c2}}}_{\texttt{C1}}$$

Don Mills

The Results, failure condition

```
property p_chain;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain;

ap_chain: assert property (p_chain);
```

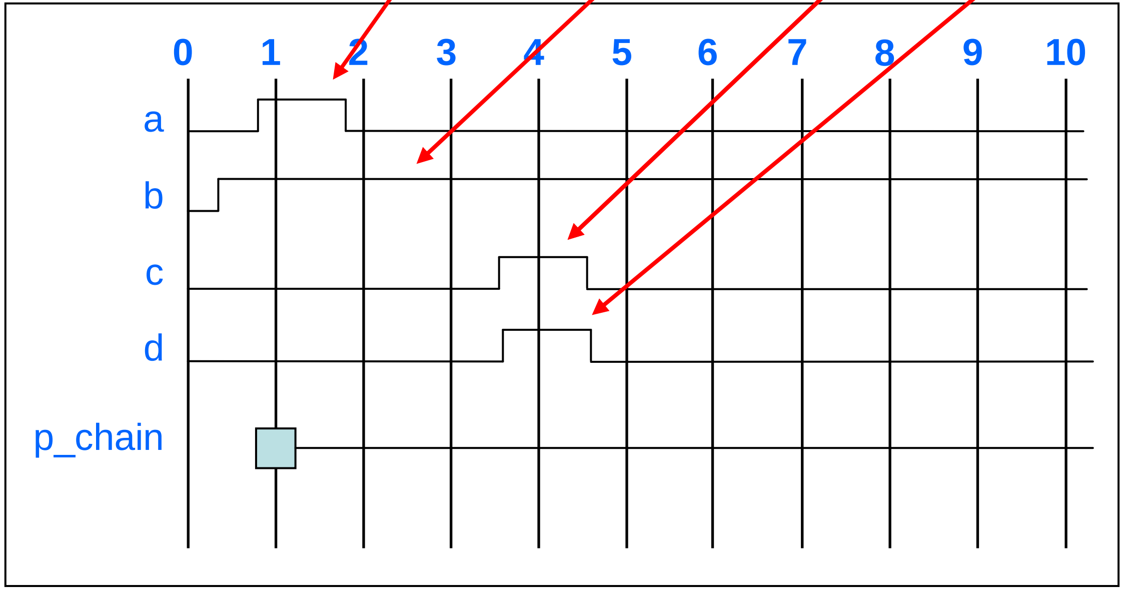


Fails at cycle 4

Don Mills

Pass attempt 1

```
property p_chain;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain;

ap_chain: assert property (p_chain);
```



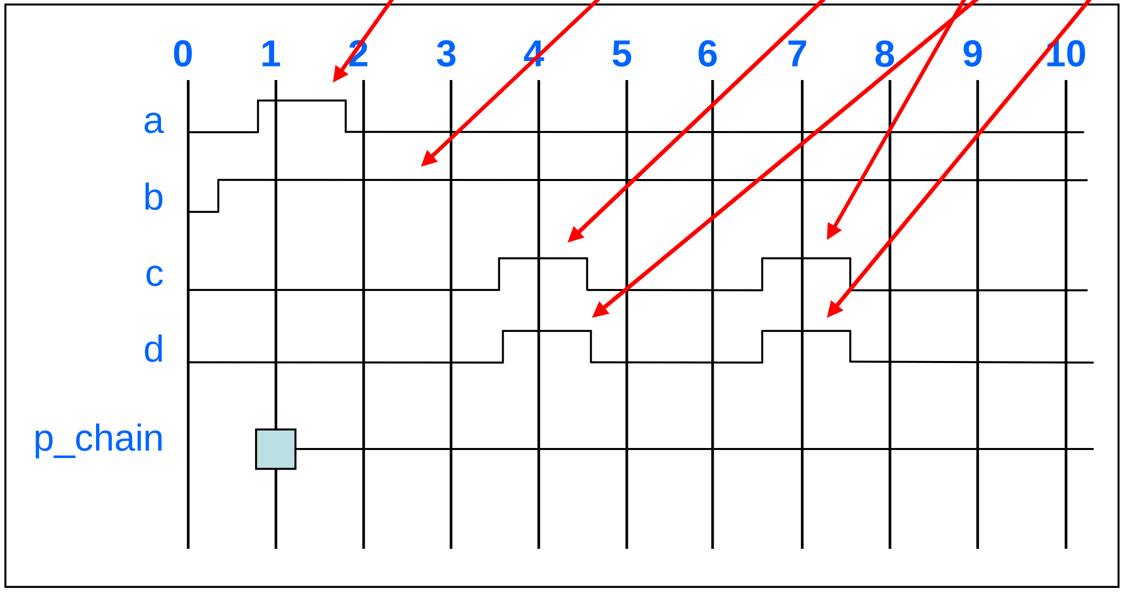
Pass or Fail at cycle 4?

Does not end with a pass at cycle 4 - BUMMER

Don Mills

Pass attempt 2

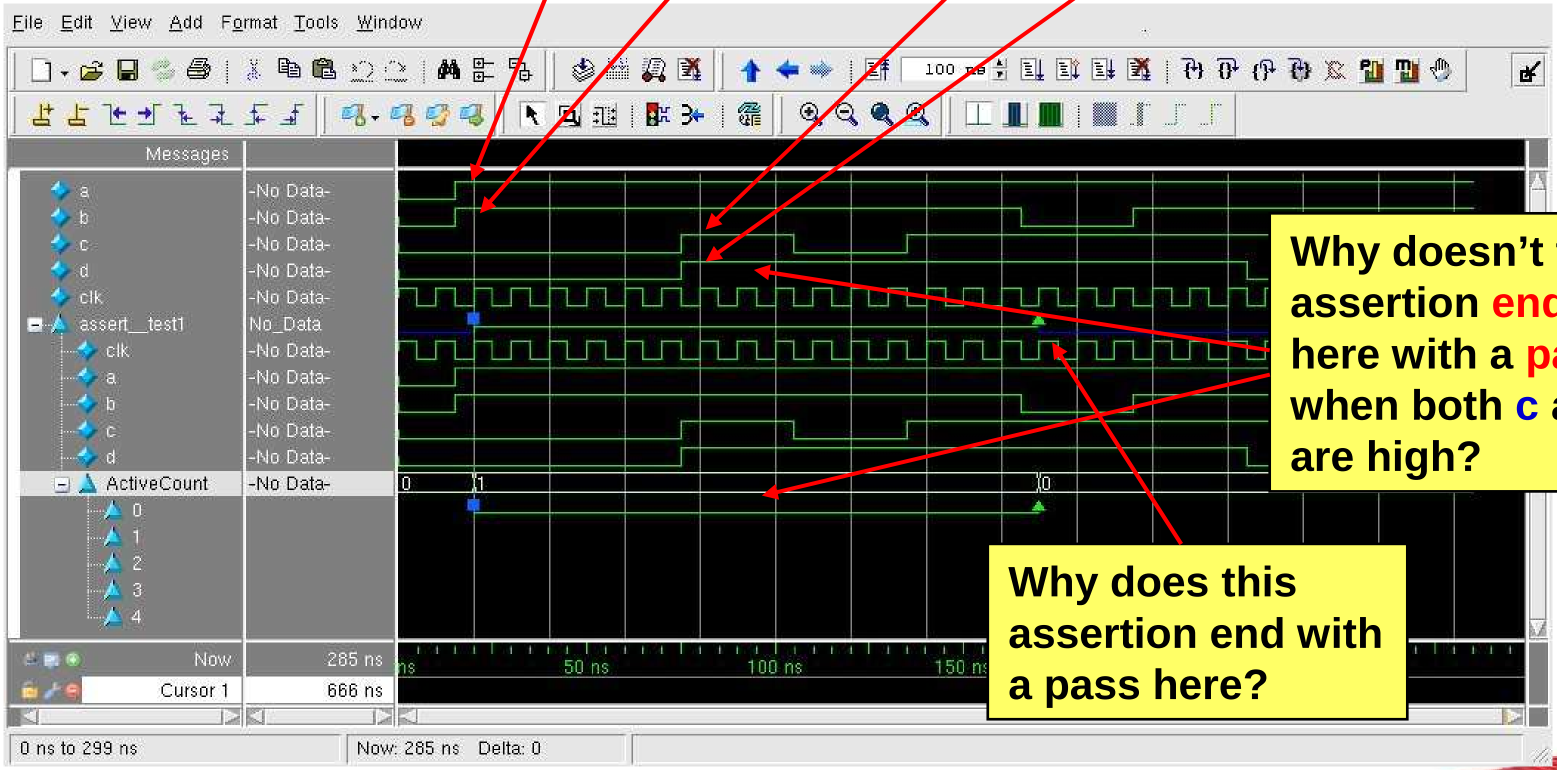
```
property p_chain;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain;

ap_chain: assert property (p_chain);
```



Does not end with a pass at cycle 4 or cycle 7

Another look…

```
property p_chain;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain;

ap_chain: assert property (p_chain);
```



Why doesn't this assertion end here with a pass when both c and d are high?

Why does this assertion end with a pass here?

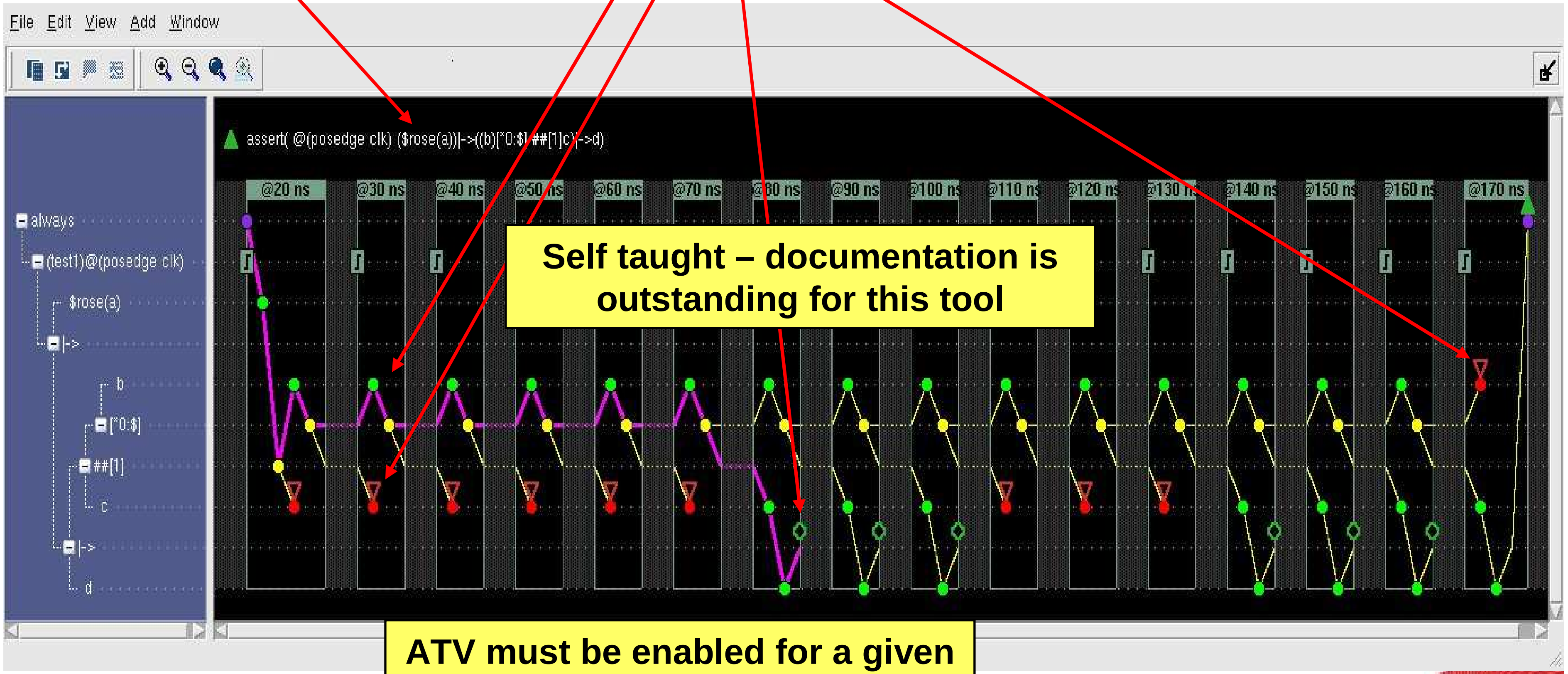
Don Mills

Assertion Thread Viewer (ATV)

Assertion statement is listed at the top of the window

Pass/Fail for tested signals is noted for each time step

Only one Assertion thread is listed at a time (slide 28)



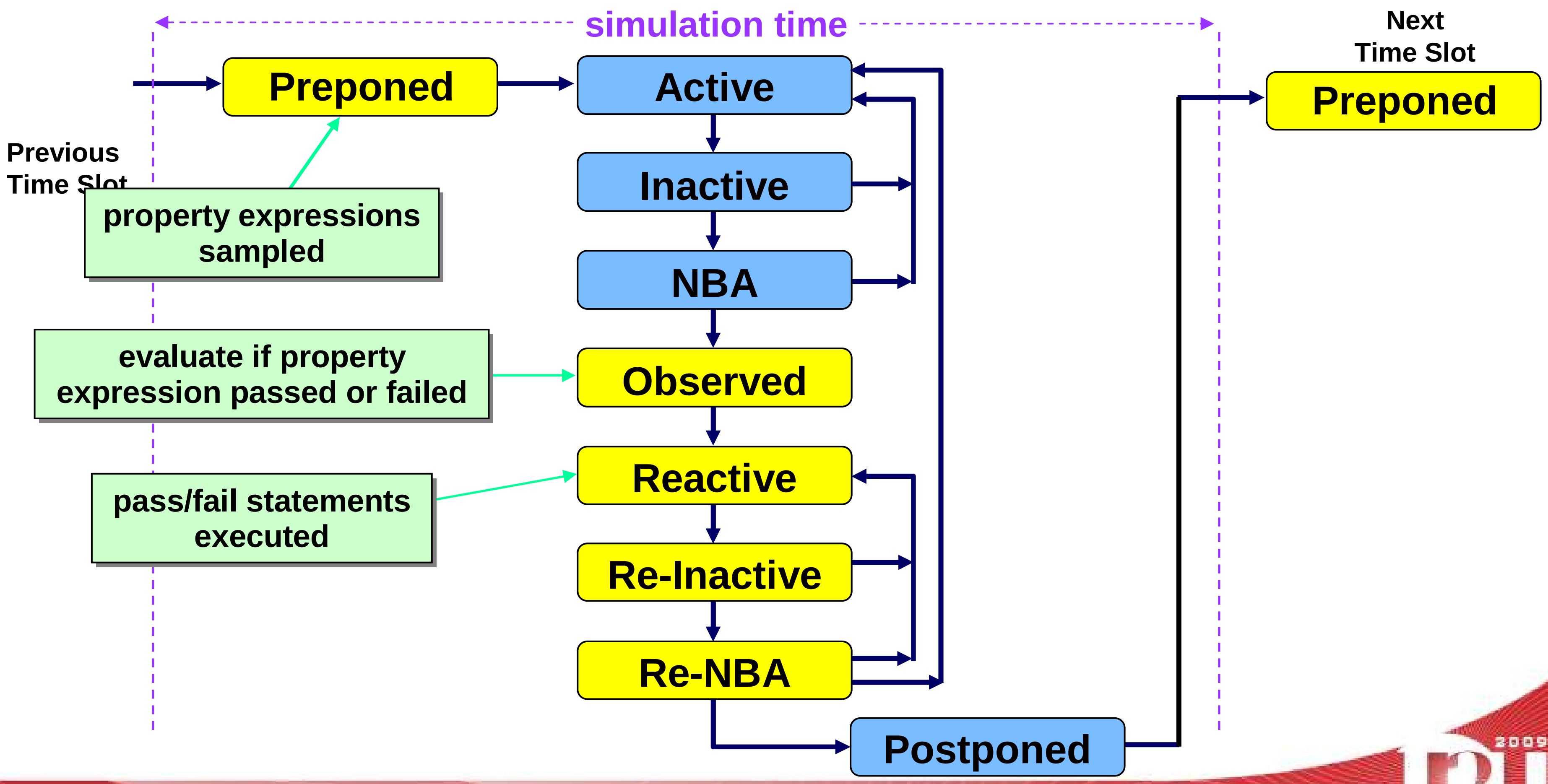
Self taught – documentation is outstanding for this tool

ATV must be enabled for a given assertion and a specific thread of that assertion prior

Concurrent Assertions Use Special Event Scheduling

Don Mills

- Concurrent assertions use special event scheduling queues:
  - Prevents race conditions with events in the design modules



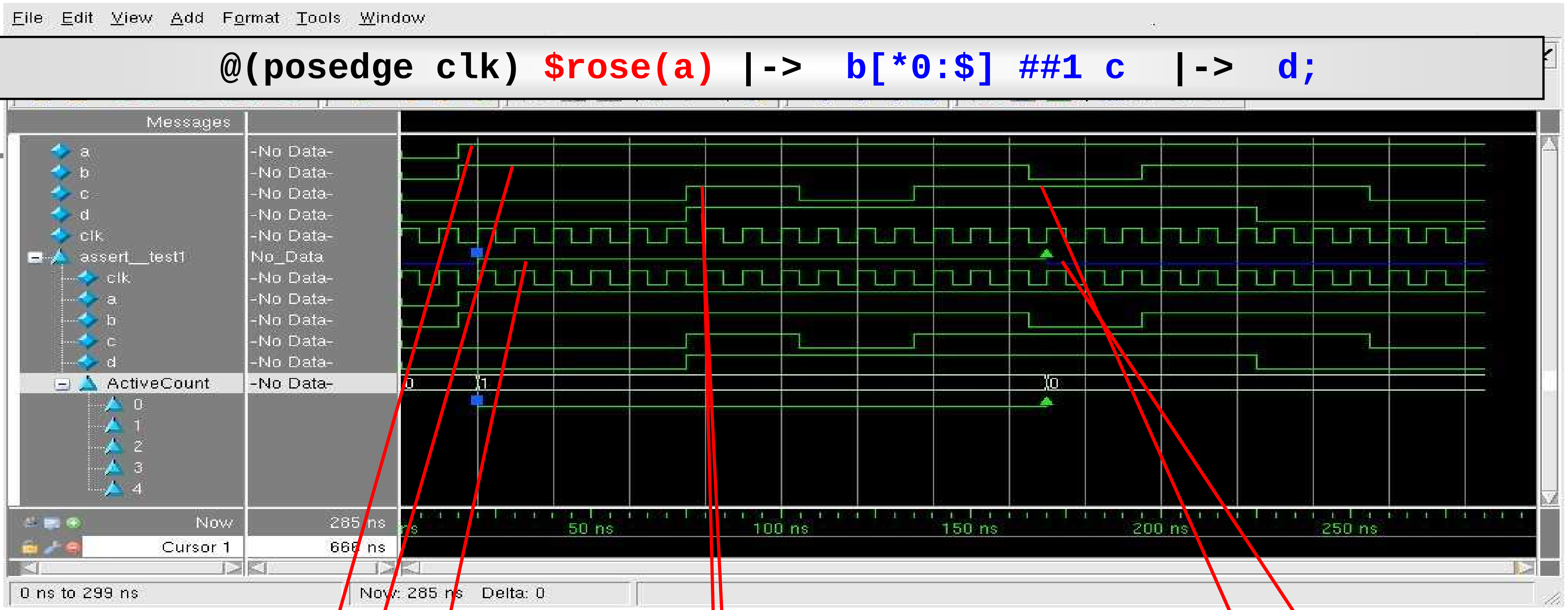
USER2USER 2009 U2U

Don Mills

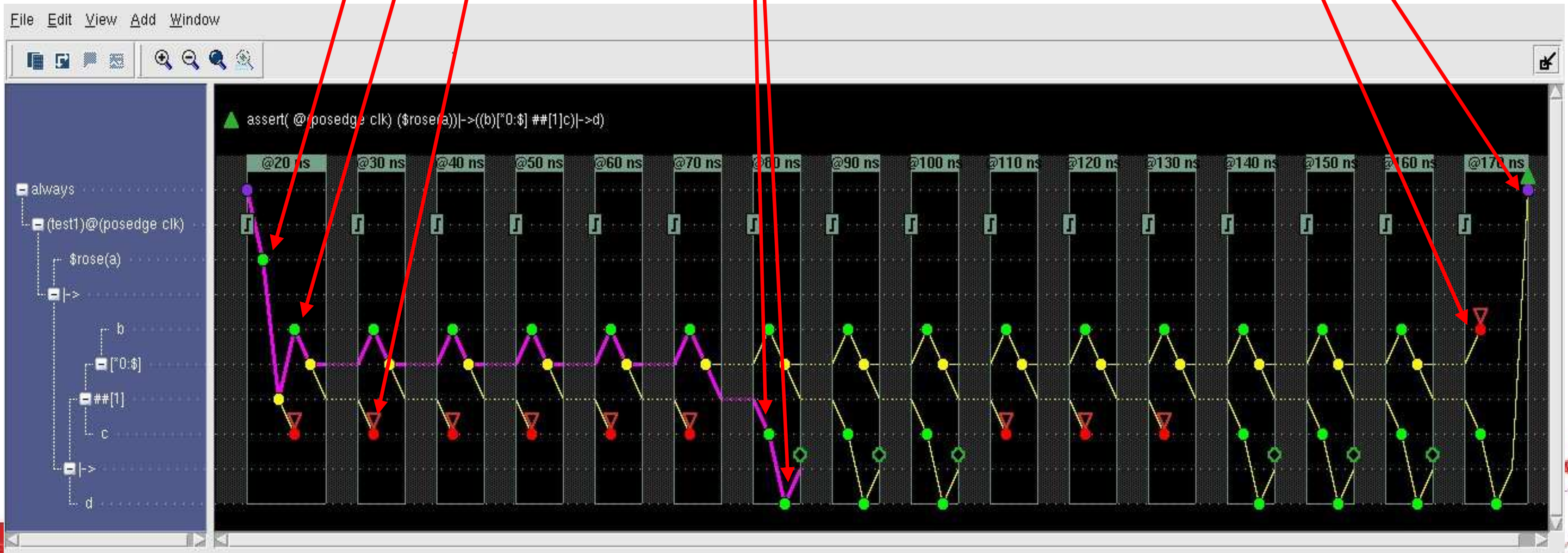
Why Use the ATV

- Concurrent Assertions are sampled at the beginning of a time step
  - If data is changing during the time step
    - Must look at the data prior to the time step
    - Like a D-FF – must look at the value of D prior the clk edge
  - ATV doesn't show the value, it shows the pass/fail for each time step

```
@(posedge clk) $rose(a) |-> b[*0:$] ##1 c |-> d;
```



```
0 ns to 299 ns          Now: 285 ns    Delta: 0
```



```
assert( @(posedge clk) ($rose(a))|->((b)[*0:$] ##[1]c)|->d)
```

So what's going on????

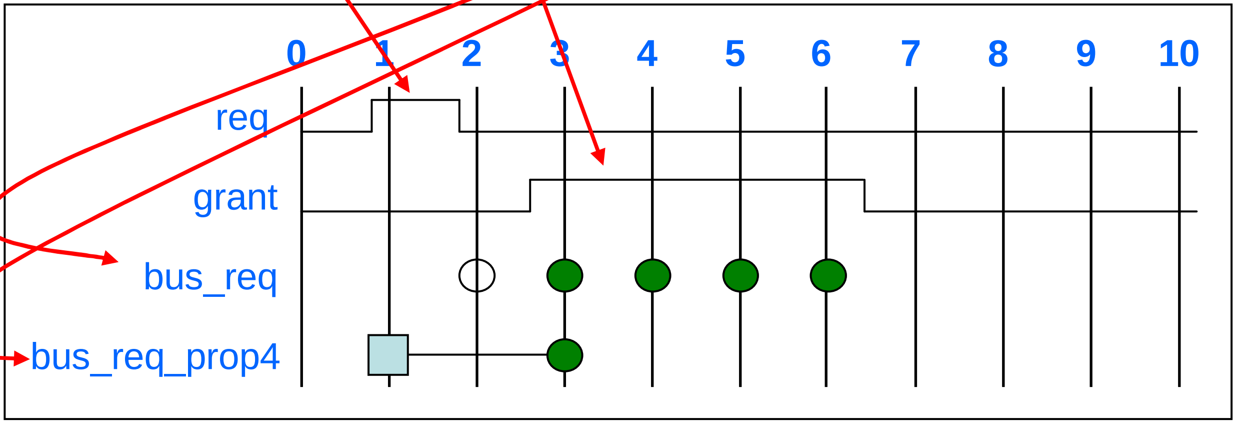
- In order to sort this out, let's look at
  - sequences and properties containing ranges
- Start with a single level implication with a
  - Range in consequent
  - Range in antecedent

Don Mills

Sequence vs. Property

```
sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop4;
  @(posedge clk) bus_req;
endproperty:bus_req_prop4

example_4: assert property (bus_req_prop4);
```

```
req ##[1:5] grant;
// equivalent to:
// req ##1 grant or
// req ##2 grant or
// req ##3 grant or
// req ##4 grant or
// req ##5 grant
```



NOTE: Only looking at the thread starting at cycle 1.

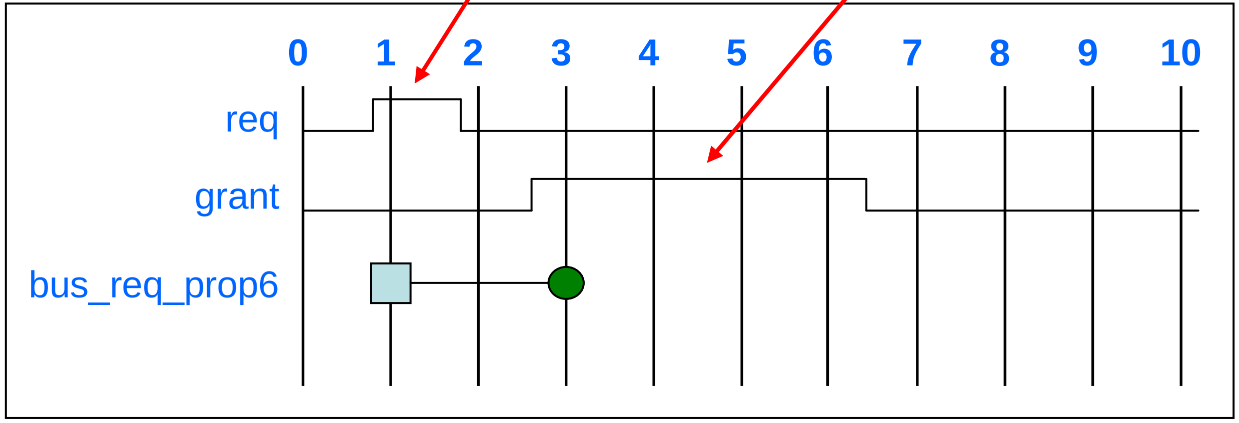
Ignoring other pass/fail cycles.

Sequence – multiple end points
Property – implied *first match*

Range in Consequent – maintains the first match rule

Don Mills
MICROCHIP

```
property bus_req_prop6;
        @(posedge clk) req |-> ##[1:5] grant;
endproperty:bus_req_prop6

example_6: assert property (bus_req_prop6);
```



First matching pass condition in the consequent ends the expression

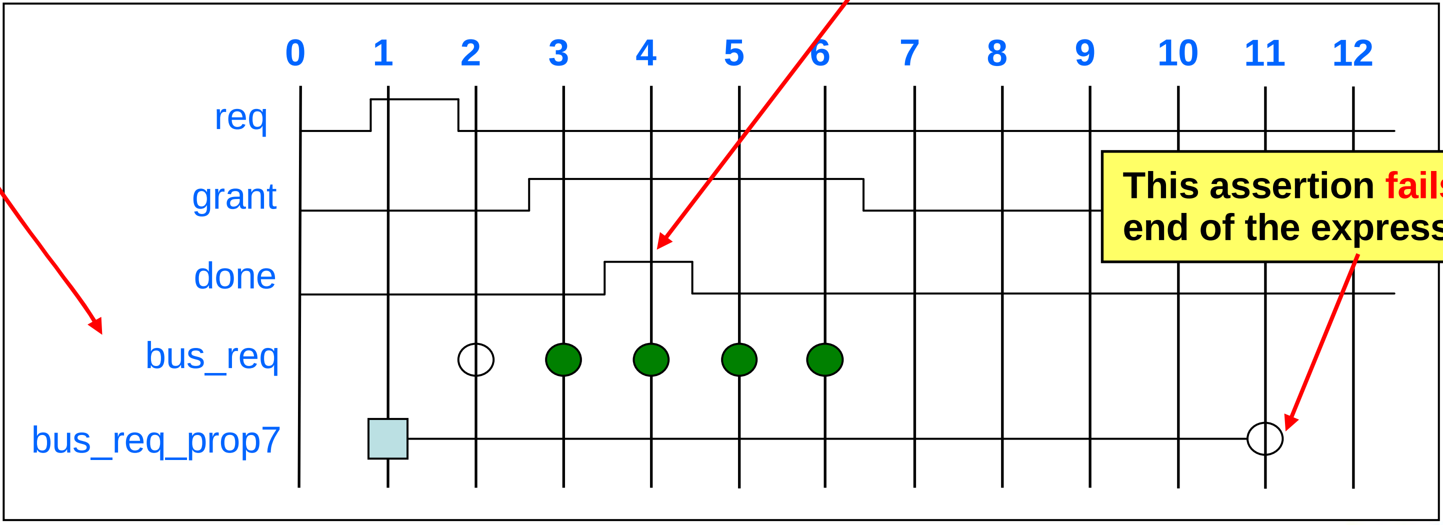
First match is implied in the consequent

Don Mills

What about a range in the antecedent?

```
sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop7;
  @(posedge clk) bus_req |->  ##[1:5] done;
endproperty:bus_req_prop7

example_7: assert property (bus_req_prop7);
```

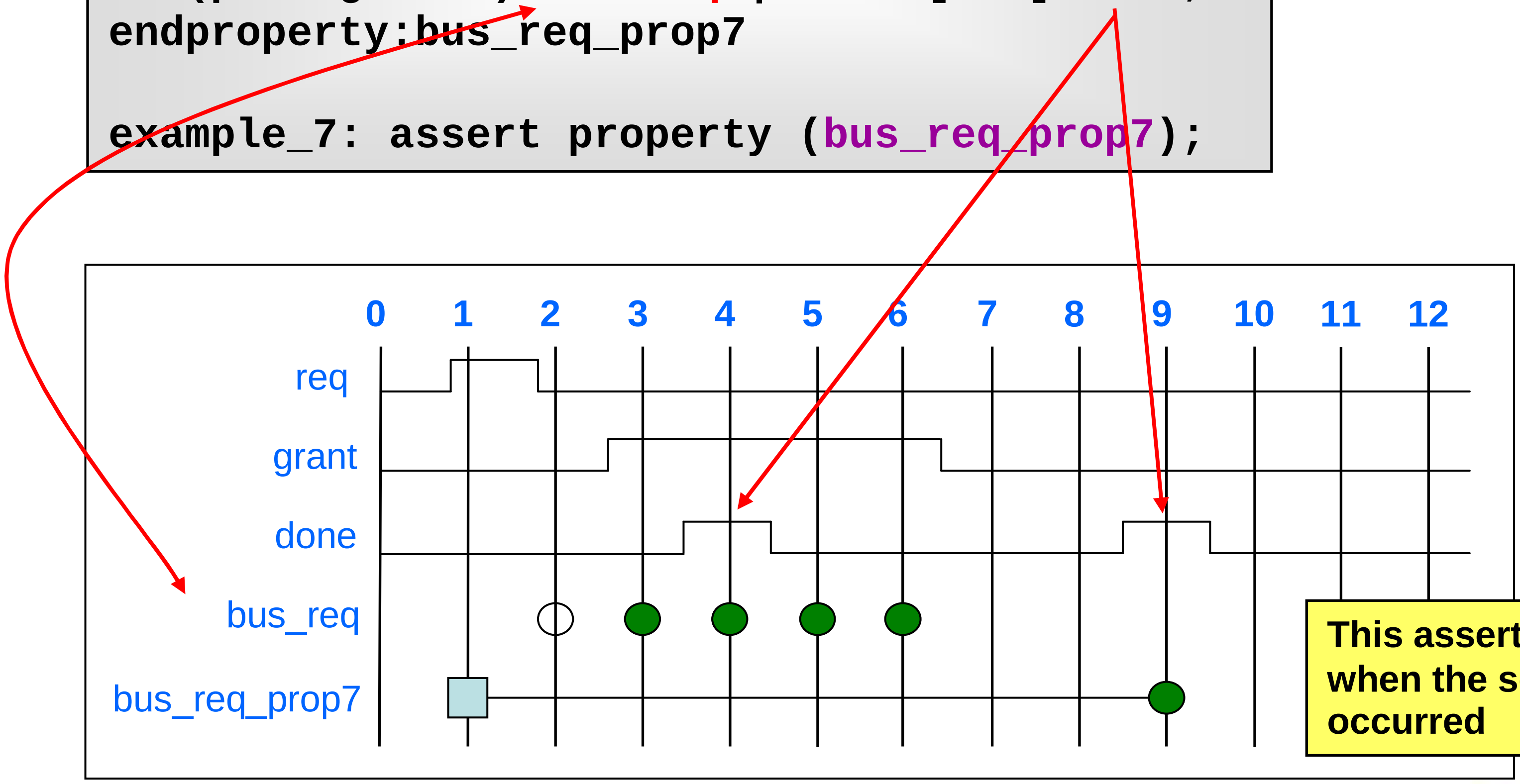
NO implied first match in the antecedent – why, why, why?



This assertion fails at the end of the expression

Range in antecedent again

```
sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop7;
  @(posedge clk) bus_req |->  ##[1:5] done;
endproperty:bus_req_prop7

example_7: assert property (bus_req_prop7);
```



This assertion passes when the second done occurred

The Problem is…
The Solution is…

Don Mills

- For antecedents with ranges
    - Every passing condition

And

-
    - Every possible future passing condition

must have a passing consequent for the property expression to PASS

- In other words – no “first match” is applied to the antecedent
- No “first match” is applied to the whole implication expression

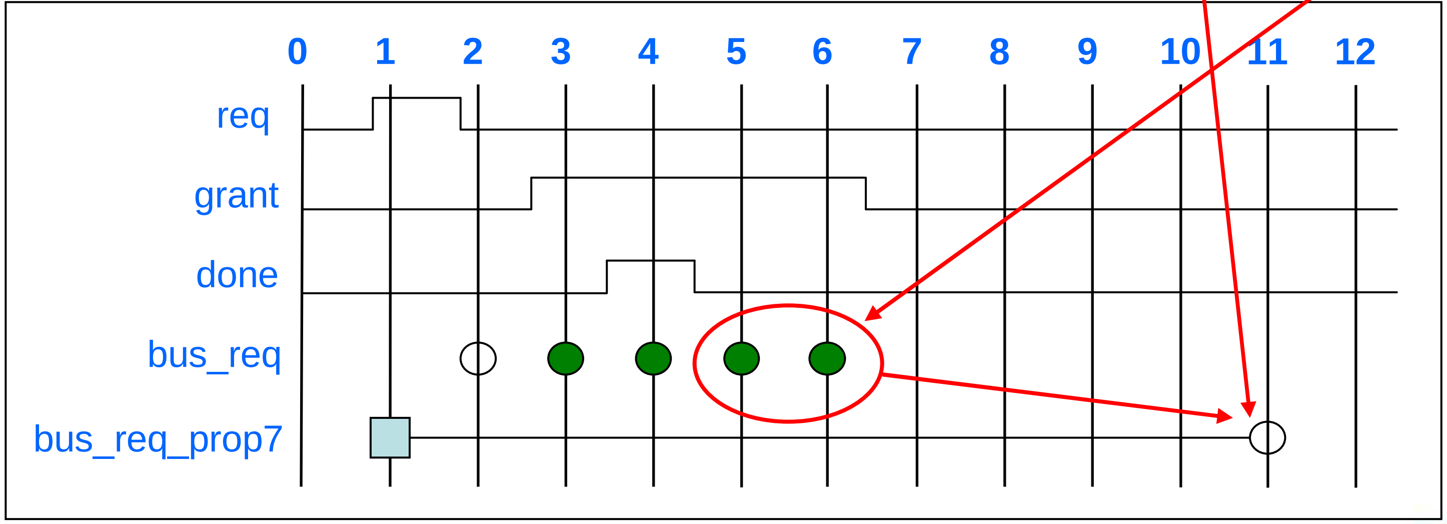
Don Mills

Range in the antecedent again… Why does this fail…

```
sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop7;
  @(posedge clk) bus_req |->  ##[1:5] done;
endproperty:bus_req_prop7

example_7: assert property (bus_req_prop7);
```

This assertion fails at the end of the expression

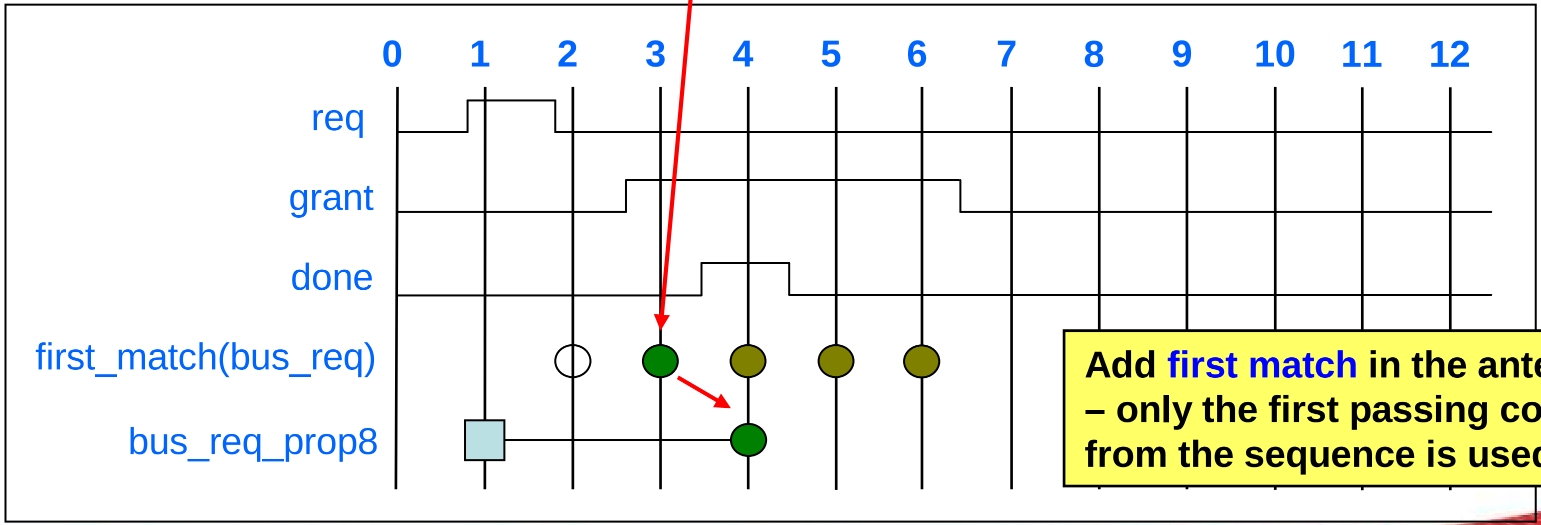
WHY:
the passing antecedent conditions at cycles 5 & 6 do not have a passing consequent



Don Mills

The FIX
Use first match on antecedent

```
sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop8;
  @(posedge clk) first_match(bus_req) |->  ##[1:5] done;
endproperty:bus_req_prop8

example_8: assert property (bus_req_prop8);
```

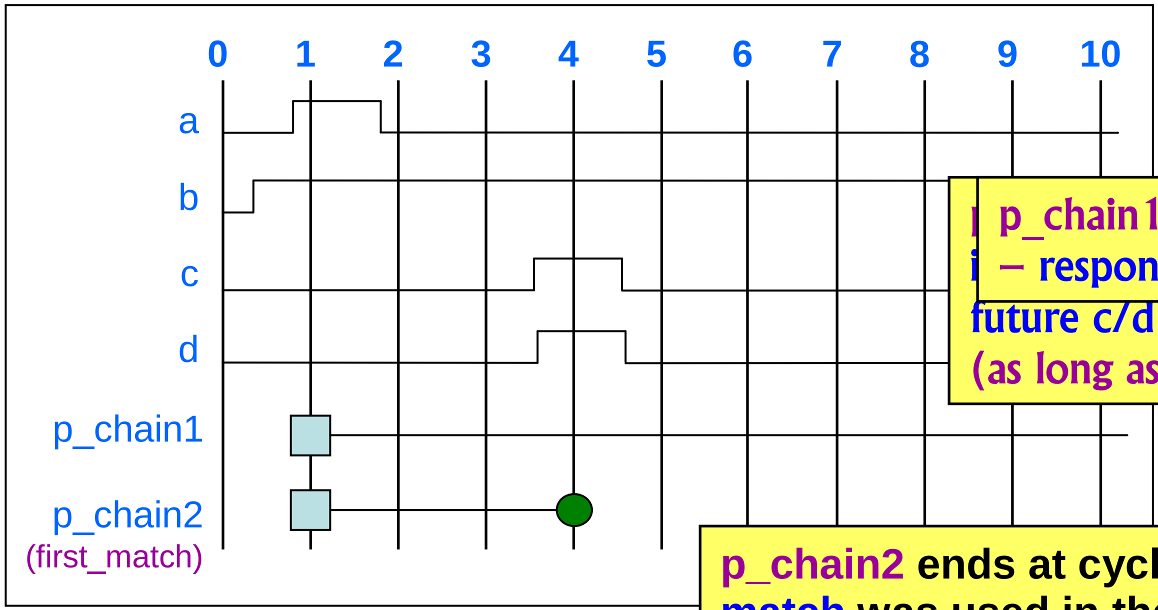


Add **first match** in the antecedent – only the first passing condition from the sequence is used.

Don Mills

Infinite upper bound range in antecedent

```
property p_chain1;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain1;

property p_chain2;
  @(posedge clk) $rose(a) |-> first_match( b[*0:$] ##1 c)  |->  d;
endproperty:p_chain2;
```



Remember - no implied first match in the antecedent

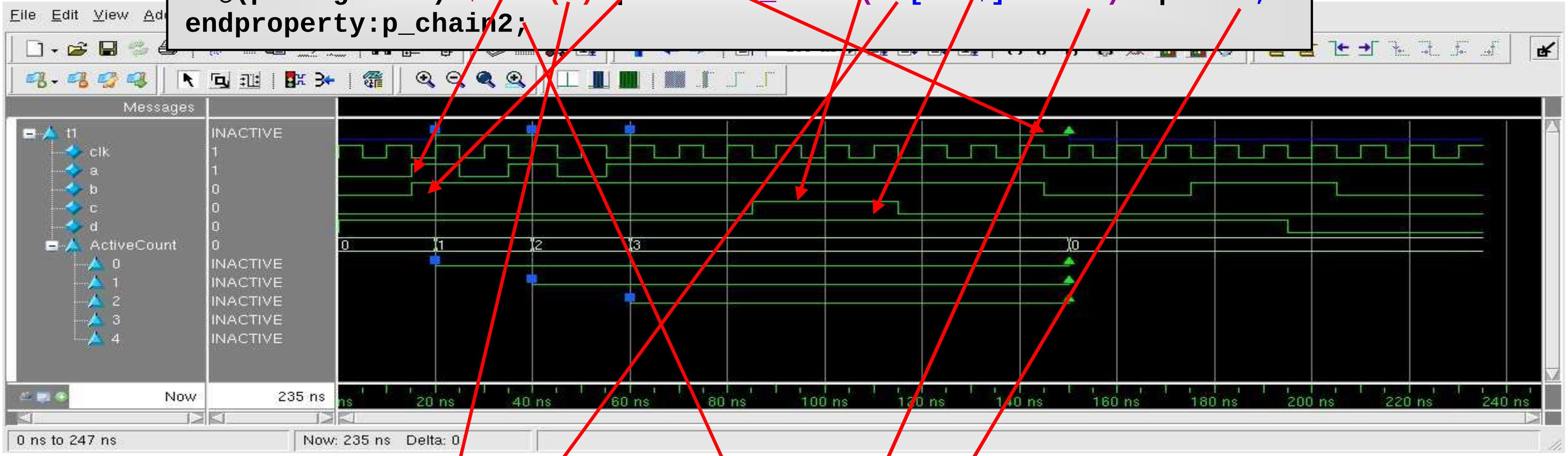
p_chain1 could be used for error checking – response when failure occurs

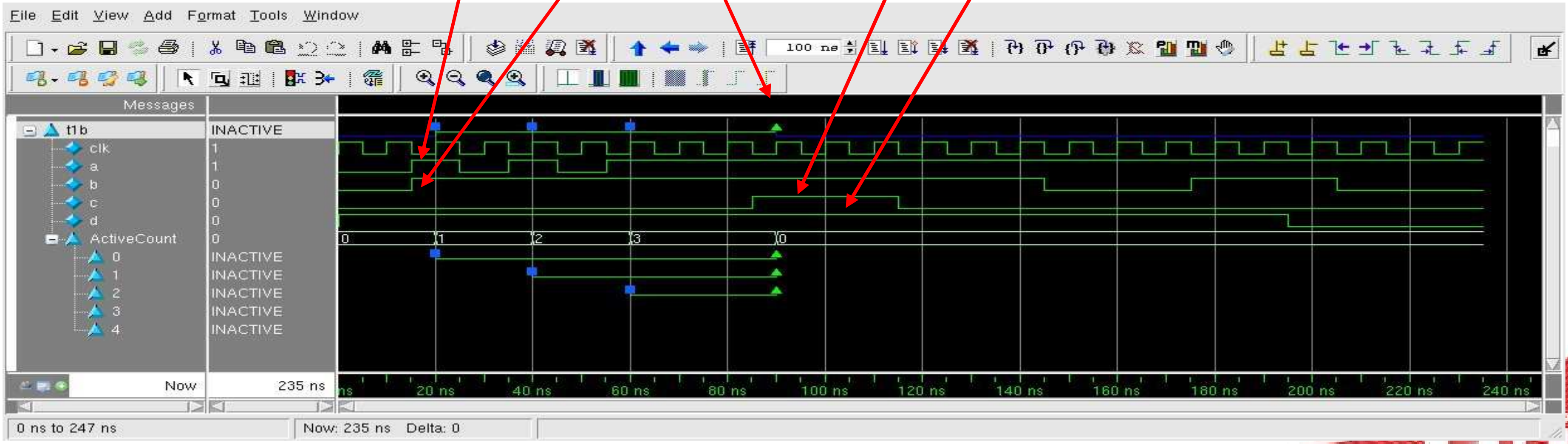
p
i
future c/d "pass" conditions
(as long as b is true)

p_chain2 ends at cycle 4 because a first match was used in the antecedent

Don Mills

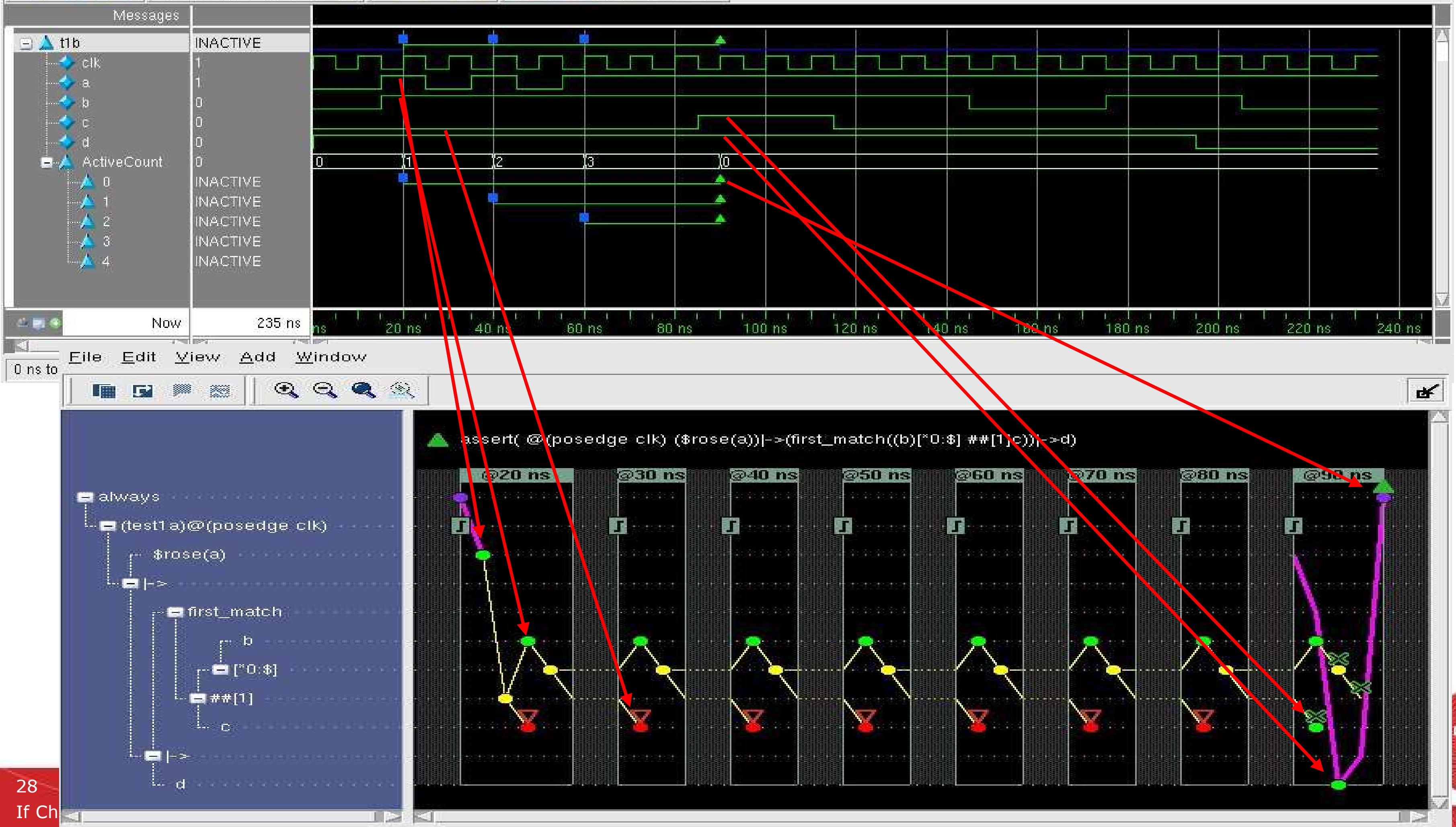
```
property p_chain1;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain1;

property p_chain2;
  @(posedge clk) $rose(a) |-> first_match( b[*0:$] ##1 c)  |->  d;
endproperty:p_chain2;
```





Firstmatch Graphics

```
property p_chain2;
  @(posedge clk) $rose(a) |-> first_match( b[*0:$] ##1 c)  |->  d;
endproperty:p_chain2;
```



Don Mills

What about vacuous results?

$$\underbrace{\texttt{\$rose(a)}}_{\texttt{A1}}\ \texttt{|->}\ \underbrace{\underbrace{\texttt{b[*0:\$] \#\#1 c}}_{\texttt{a2}}\ \texttt{|->}\ \underbrace{\texttt{d;}}_{\texttt{c2}}}_{\texttt{C1}}$$

A1 fails = vacuous success
A1 passes, a2 fails = vacuous success

Both A1 and a2 must pass for a non-vacuous success to be possible

How do I model a vacuous success from A1 only?

Don Mills

$$\underbrace{\texttt{\$rose(a)}}_{A1}\ \texttt{|->}\ \underbrace{\underbrace{\texttt{b[*0:\$] \#\#1 c}}_{a2}\ \texttt{|->}\ \underbrace{\texttt{d;}}_{c2}}_{C1}$$

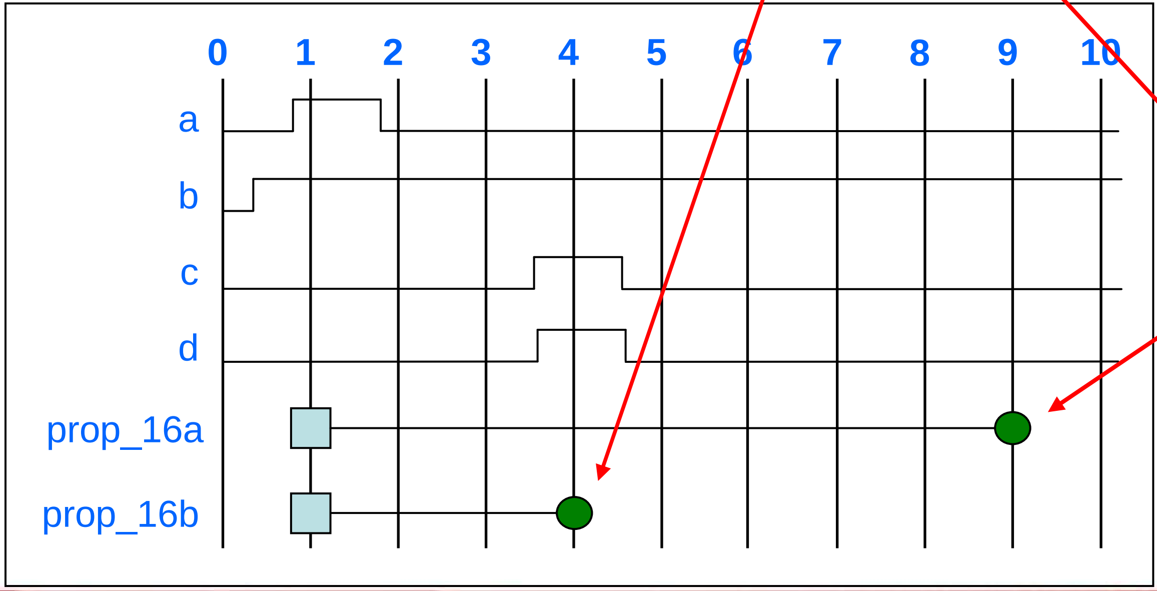
```
property prop_19a;
  @(posedge clk) A1 |-> a2 |-> c2 ;
endproperty:prop_19a

property prop_19b;
  @(posedge clk) A1 |-> a2;
endproperty:prop_19b

property prop_19c;
  @(posedge clk) prop_19a and prop_19b;
endproperty:prop_19c
```

Vacuous for A1 only requires and'ing two properties together

What if the upper bound was fixed and not infinity?

Don Mills

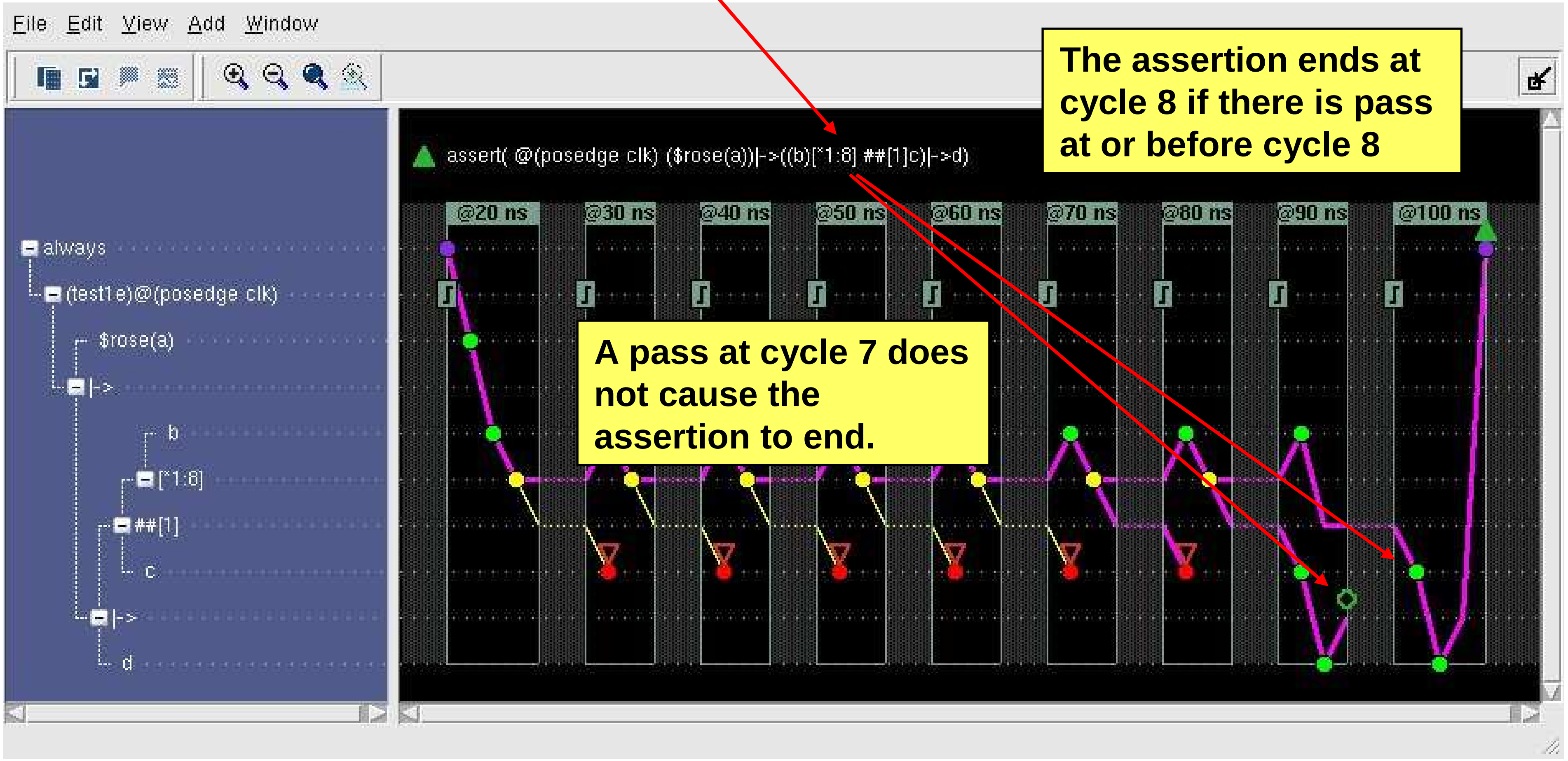
```
property prop_16a;
  int v_cnt;
  @(posedge clk) $rose(a) |-> b[*0:8] ##1 c |-> d;
endproperty:prop_16a

property prop_16b;
  int v_cnt;
  @(posedge clk) $rose(a) |-> first_match(b[*0:8] ##1 c) |-> d;
endproperty:prop_16b
```



A fixed upper bound can end with a pass once the upper bound is reached

Don Mills

Fixed upper bound



assert(@(posedge clk) ($rose(a))|->((b)[*1:8] ##[1]c)|->d)

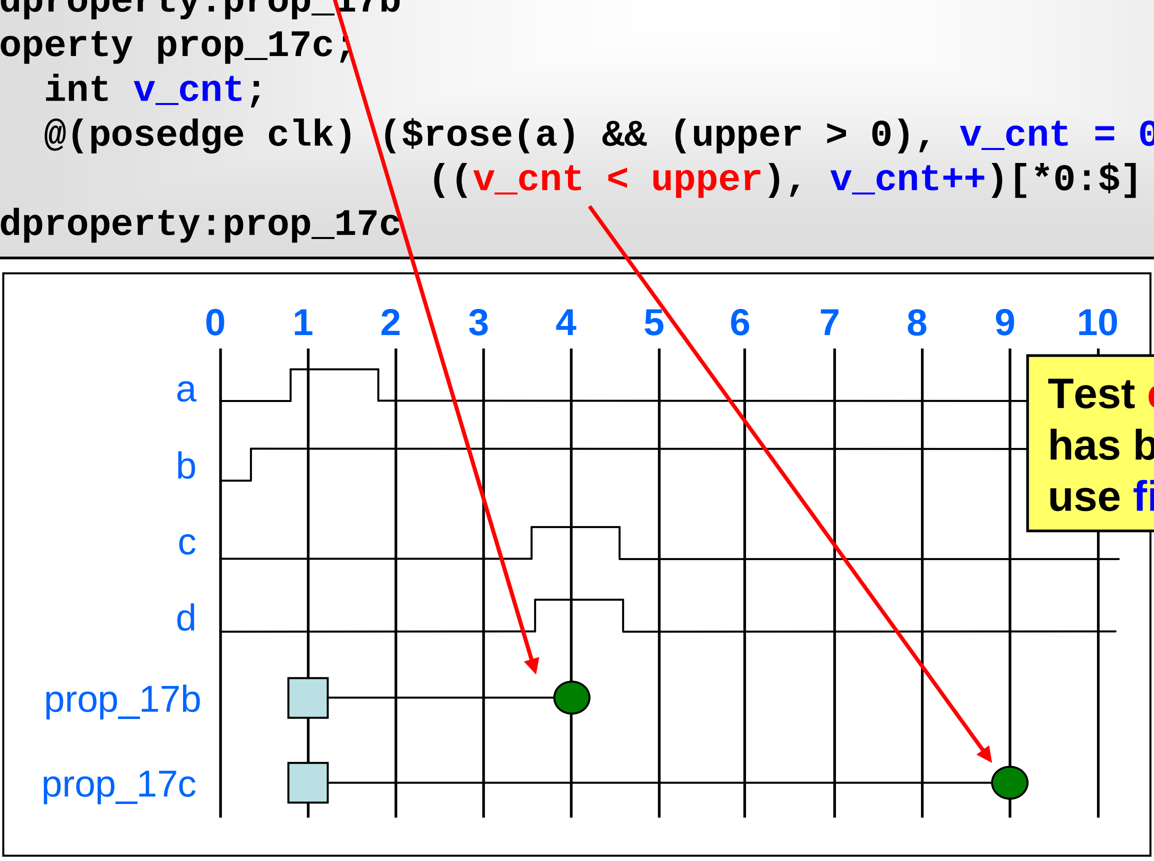
The assertion ends at cycle 8 if there is pass at or before cycle 8

A pass at cycle 7 does not cause the assertion to end.

Don Mills

Variable upper range limit

```
module foo;  int upper = 8;
property prop_17b;
    int v_cnt;
    @(posedge clk) ($rose(a) && (upper > 0), v_cnt = 0) |->
        first_match(((v_cnt < upper), v_cnt++)[*0:$] ##1 c) |-> d;
endproperty:prop_17b
property prop_17c;
    int v_cnt;
    @(posedge clk) ($rose(a) && (upper > 0), v_cnt = 0) |->
                    ((v_cnt < upper), v_cnt++)[*0:$] ##1 c |-> d;
endproperty:prop_17c
```



Test c/d until the upper count has been reached. Can still use first match if desired.

Variation – only test when the variable upper limit is reached

Don Mills

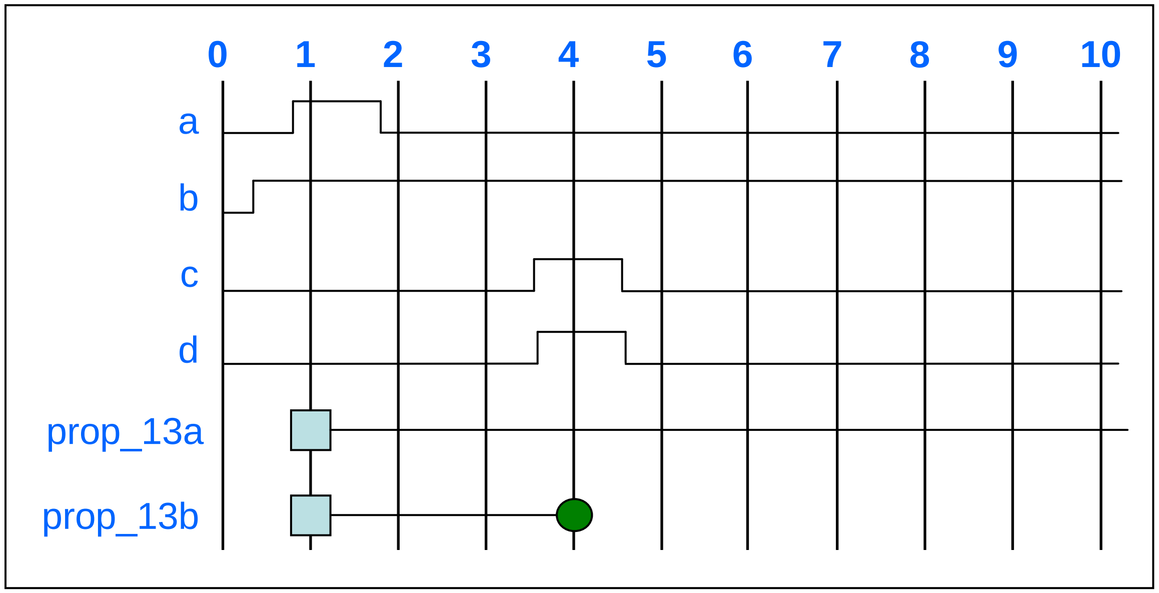
```
module foo;
  bit a, c, d, clk;
  int upper;

  property prop_18(cnt);
    int v_cnt;
    @(posedge clk) ($rose(a) && (cnt > 0), v_cnt = 0) |->
        ((v_cnt < cnt), v_cnt++)[*0:$] ##1
        (v_cnt == cnt) ##0 c |-> d;
  endproperty:prop_18

  ap18: assert property (prop_18(upper));

  initial begin
    upper = 8;
```

Only test c/d when (v_cnt == cnt)

Another way to model chained implication is using fusion (##0)

Don Mills

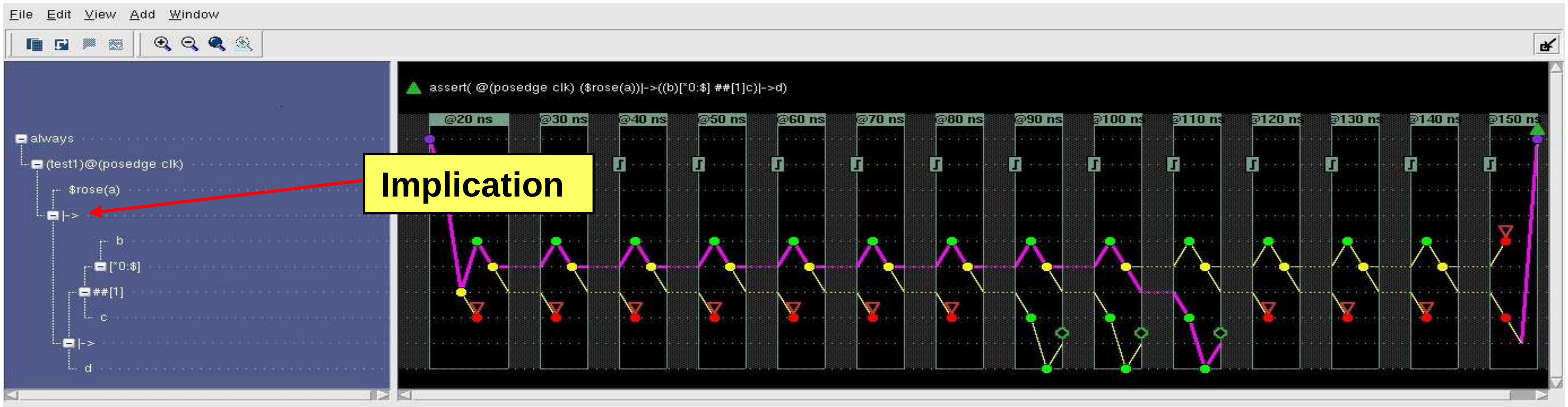
```
property prop_13a;
  int v_cnt;
  @(posedge clk) $rose(a) ##0 (b[*0:$] ##1 c) |-> d;
endproperty:prop_13a

property prop_13b;
  int v_cnt;
  @(posedge clk) $rose(a) ##0 first_match(b[*0:$] ##1 c) |-> d;
endproperty:prop_13b
```



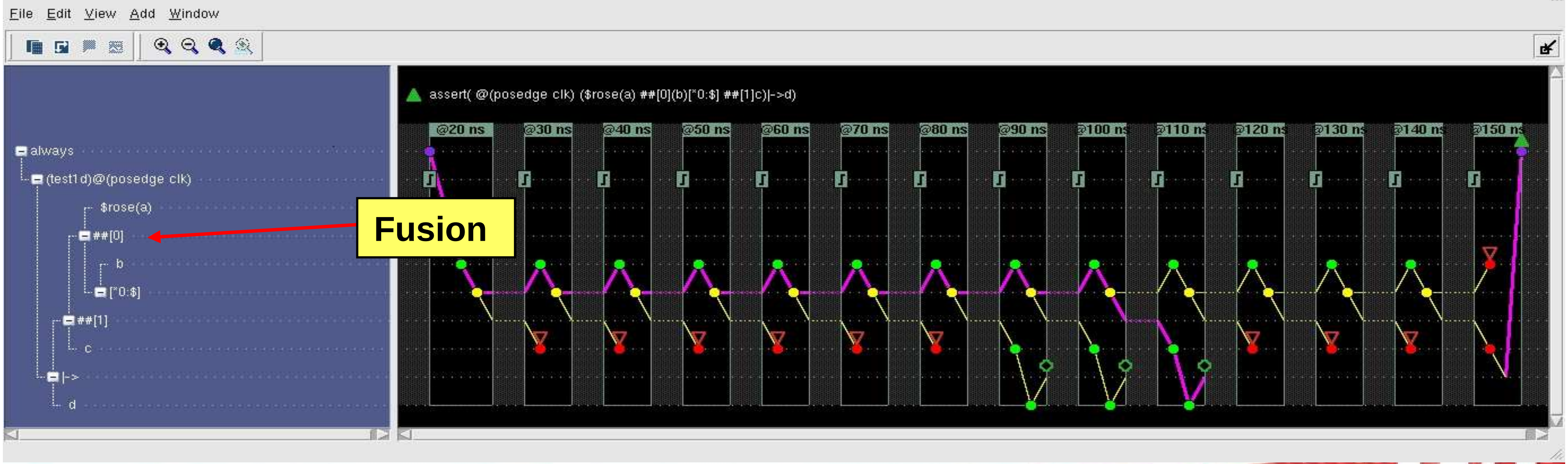
Replacing the first implication operator with a fusion gives identical results

Don Mills

Implication vs. Fusion – same results



```
assert( @(posedge clk) ($rose(a))|->((b)[*0:$] ##[1]c)|->d)
```



```
assert( @(posedge clk) ($rose(a) ##[0](b)[*0:$] ##[1]c)|->d)
```

Summary

- An implication will not end until all possible antecedent “passes” have tested with a passing consequent
- An implication with a range in the antecedent ends when
  - A passing antecedent has a failing consequent
  - The end of the range occurs
  - first_match is used on the antecedent and the consequent passes.

```
`de

property p_max_cycles;
  int v_cnt;
  @(posedge clk) ($rose(start), v_cnt = 0) |->
         first_match((`TRUE, v_cnt++)[*0:$] ##1 done) |->
                        (v_cnt <= MAX);
endproperty:p_max_cycles

ap_max_cycles: assert property (p_max_cycles);
```

Range – can be either a repetition range or timing range

les between the start and the end point is less than the max allowed

Questions & Answers…

- QUESTIONS and GUESSES

U2U 2009

USER2USER