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

Don Mills
don.mills@microchip.com
mills@lcdm-eng.com

Microchip Technology Inc.
Chandler, AZ, USA

www.microchip.com

Assumptions

- You know basicly what assertions are
- You know basicly what properties are
- You know basicly what sequences are
- You know what what range repitition is

What is an Implication?

- 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
```

Why use chained implications

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

```
Psuedo 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

Don't want an assertion failure on one of the "en" conditions being false

  - Very difficult to create this conditioning with multiple properties
  - Can use a local variables between the different levels.

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 endpoint is less then the max allowed

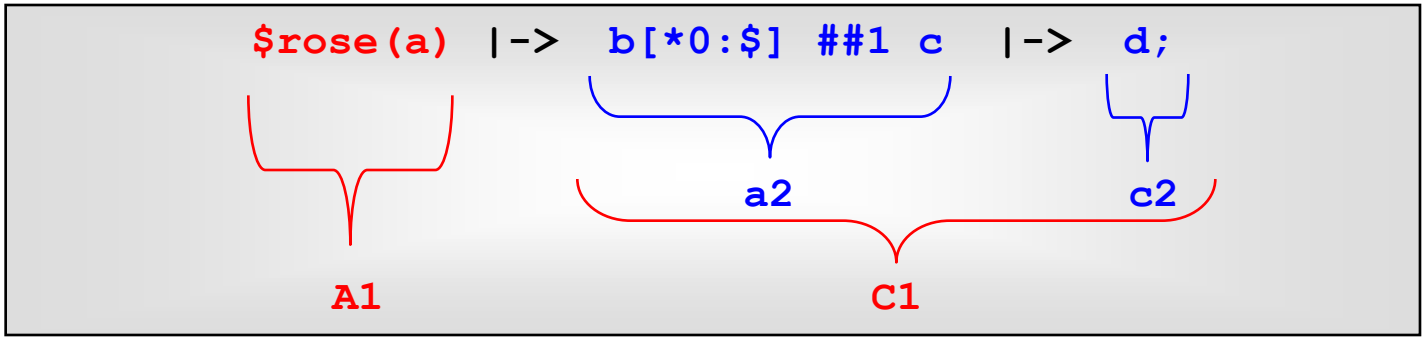
```
`define TRUE 1

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

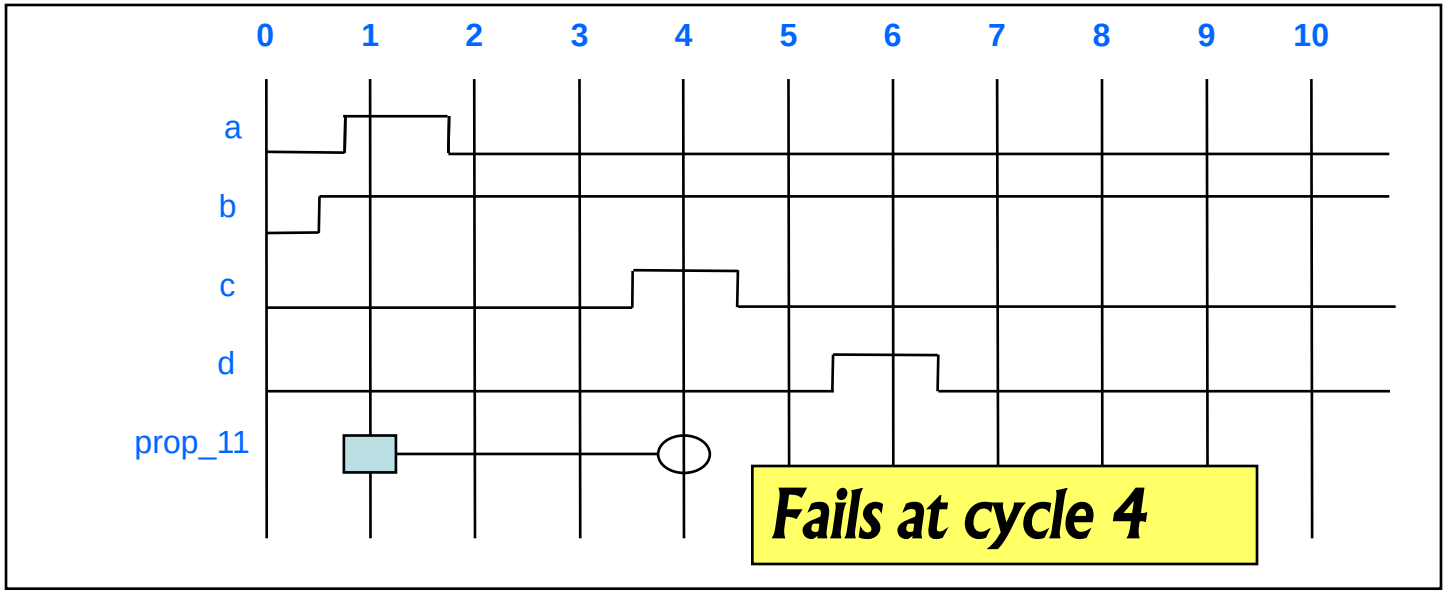
```
property p_chain;
  @(posedge clk) $rose(a) |->  b[*0:$] ##1 c  |->  d;
endproperty:p_chain;

ap_chain: assert property (p_chain);
```

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



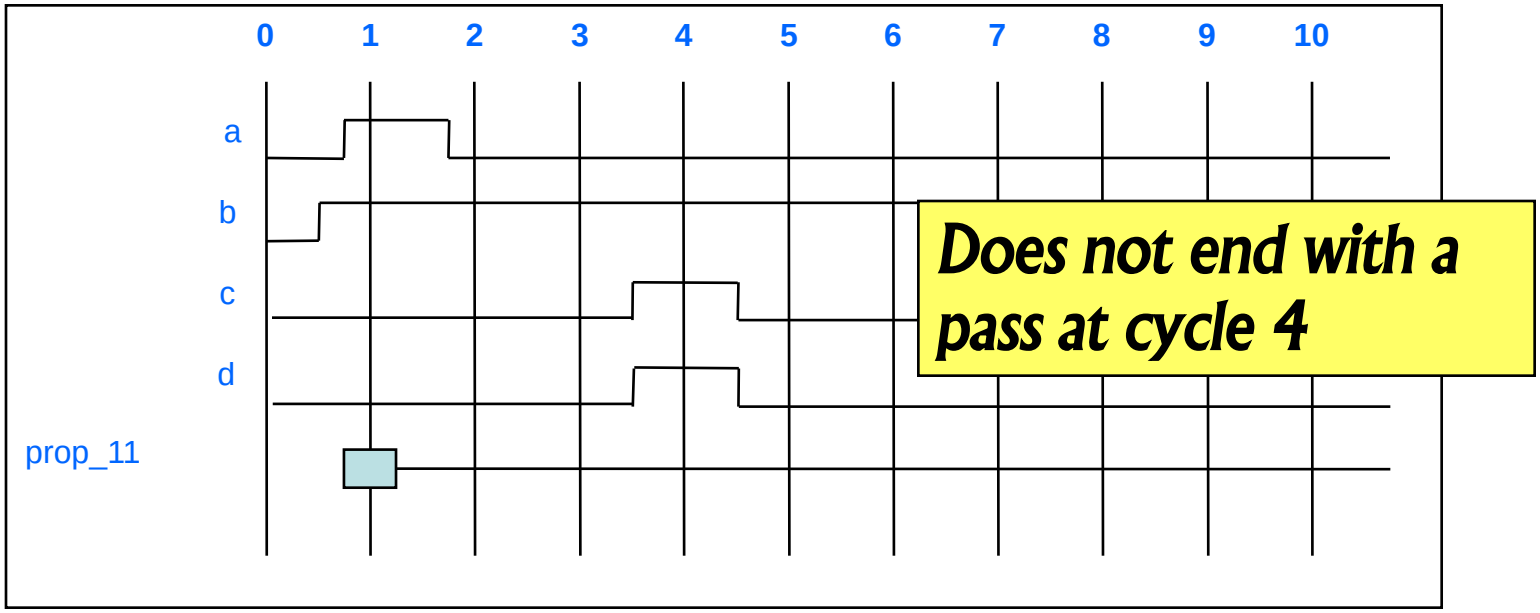
The Results, Why?

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

ap_chain: assert property (p_chain);
```



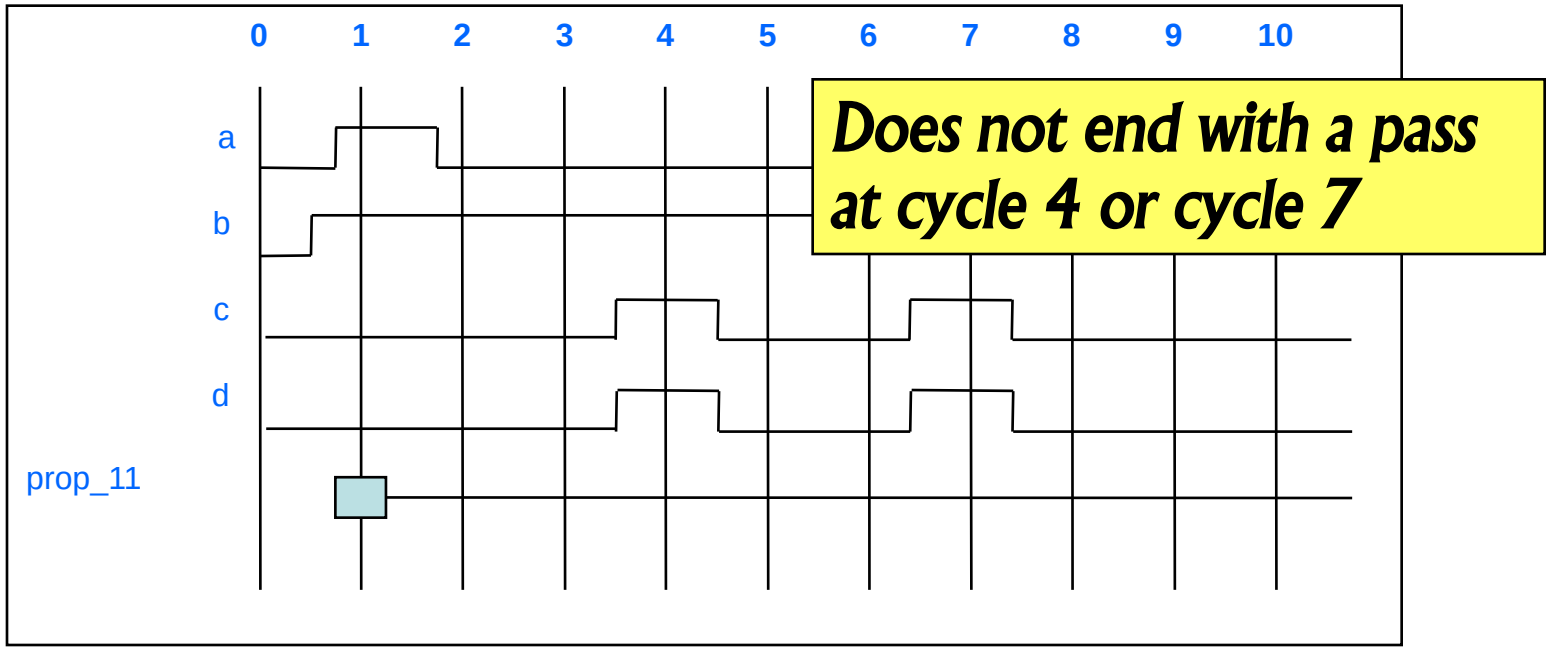
Pass attempt 1

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

ap_chain: assert property (p_chain);
```



Pass attempt 2

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

ap_chain: assert property (p_chain);
```



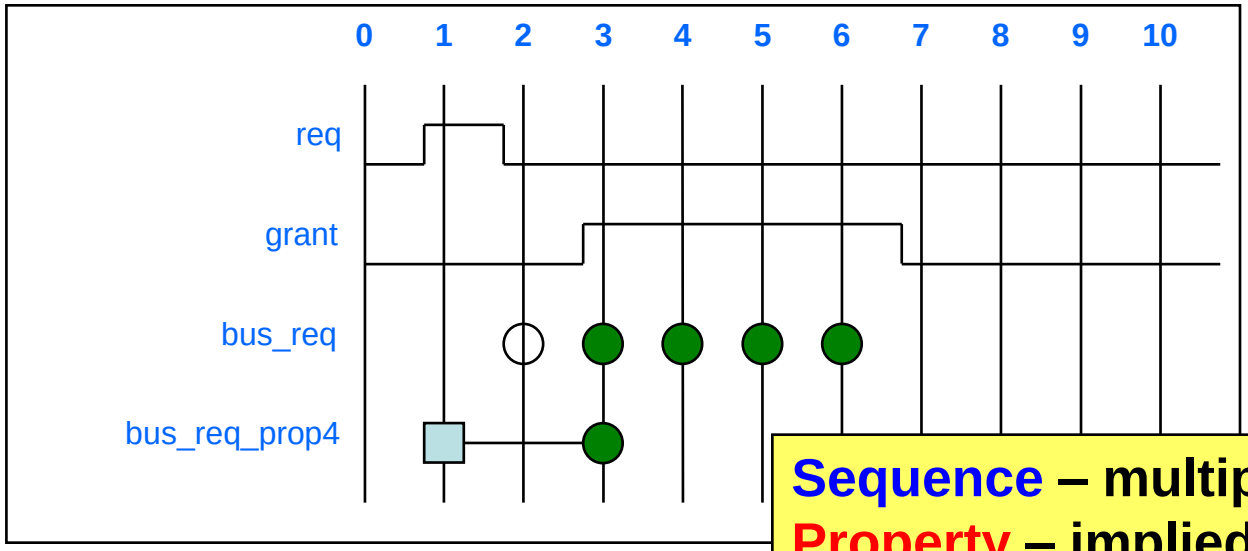
So what's going on????

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

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
```



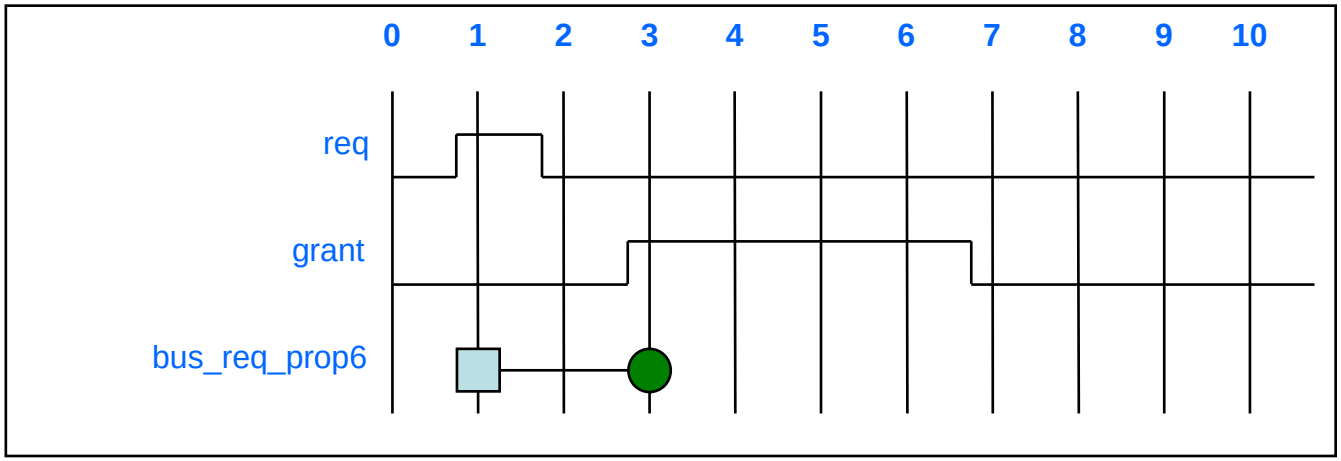
Only looking at the thread starting at cycle 1.

Ignoring other pass/fail cycles

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

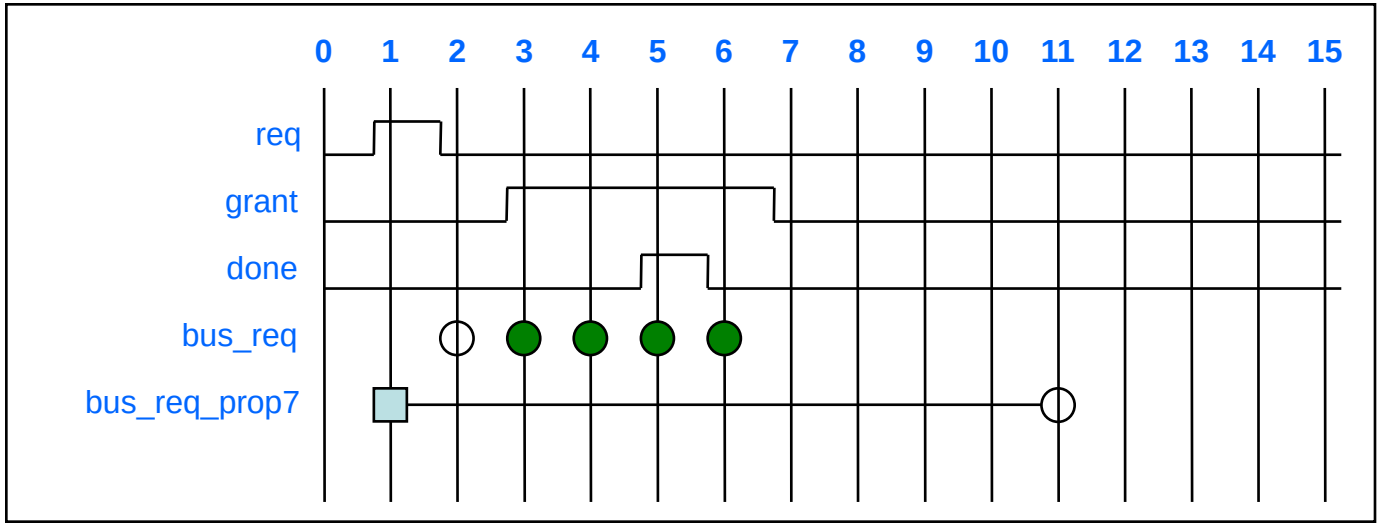
Range in Consequence – maintains the first match rule

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

example_6: assert property (bus_req_prop6);
```



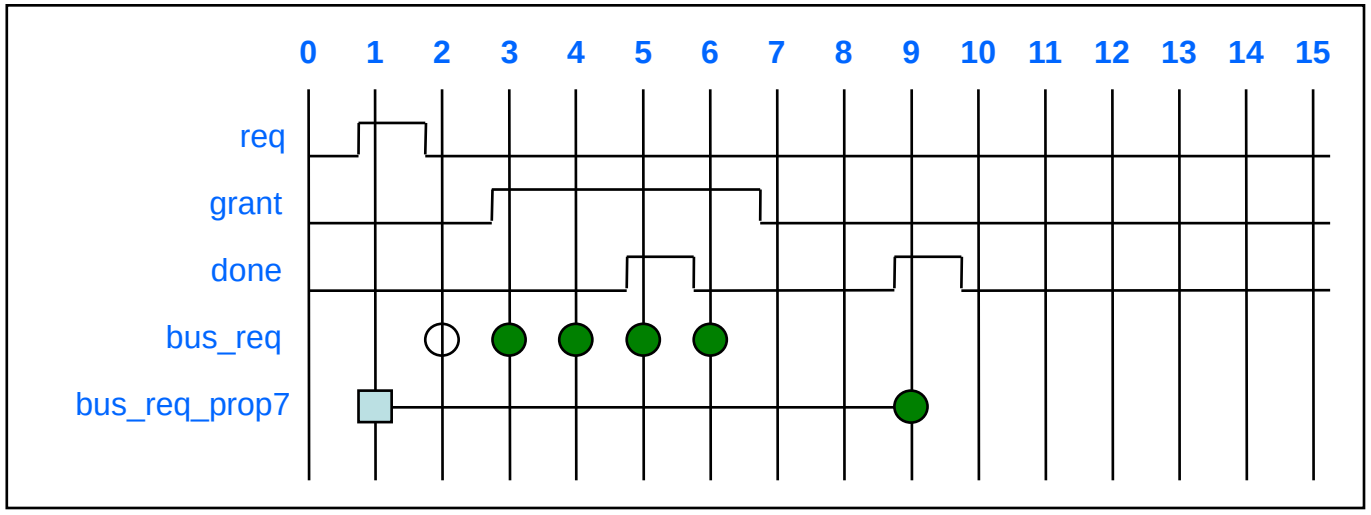
What about a range in the antecedent?

```
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);
```



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);
```



Problem is…

- Antecedents with ranges
  - Every passing condition from a range

And

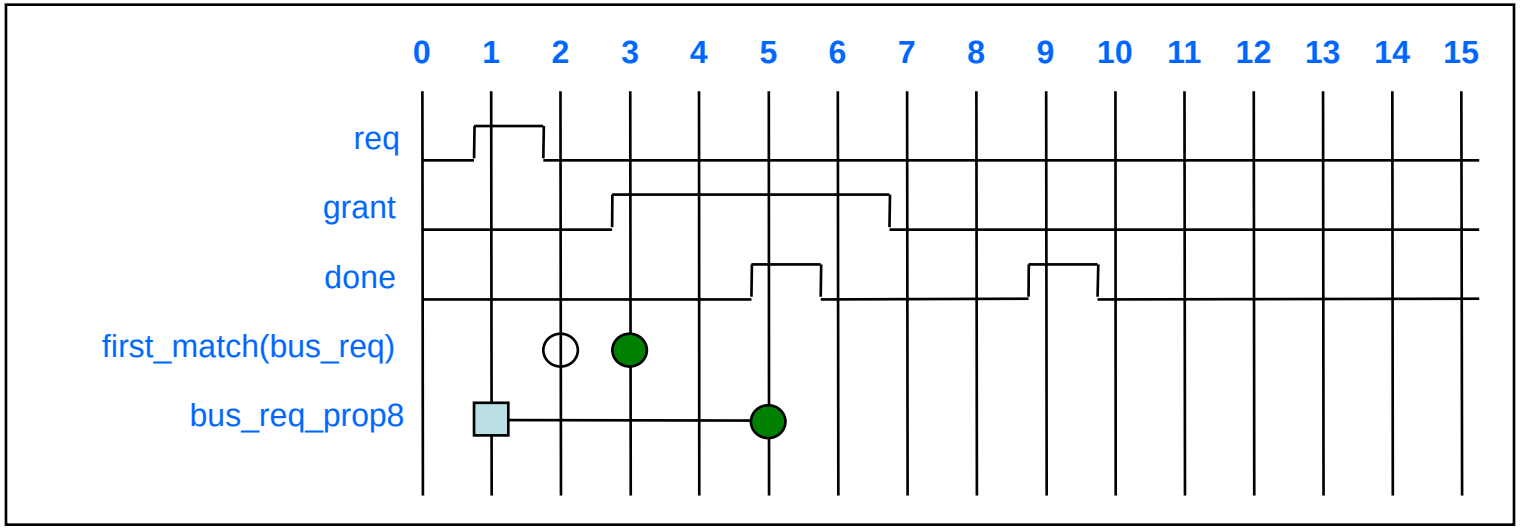
  - Every posible future passing condition from a range

must have a passing consequent for the property expression to PASS

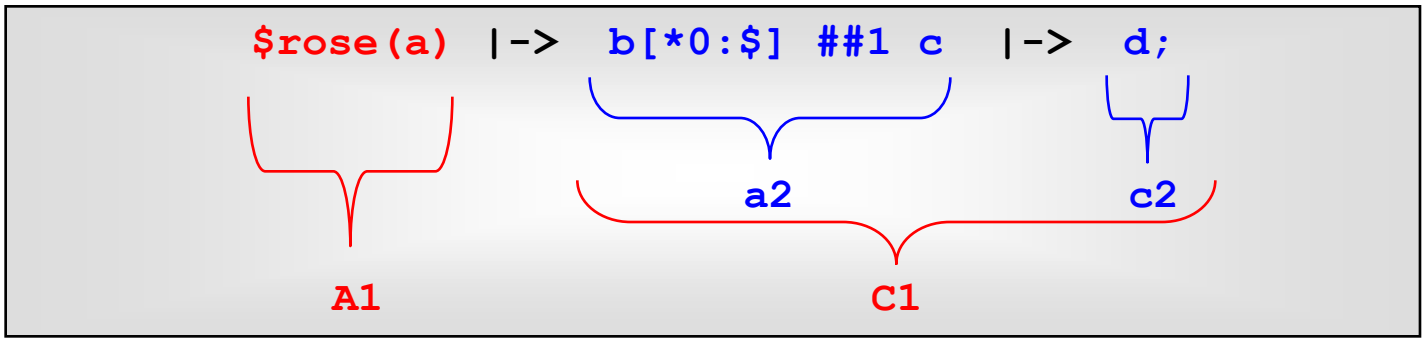
- In other words – no “first match” is applied to the antecedent
- No “first match” is applied to the whole implication expression
  - Only to the consequent

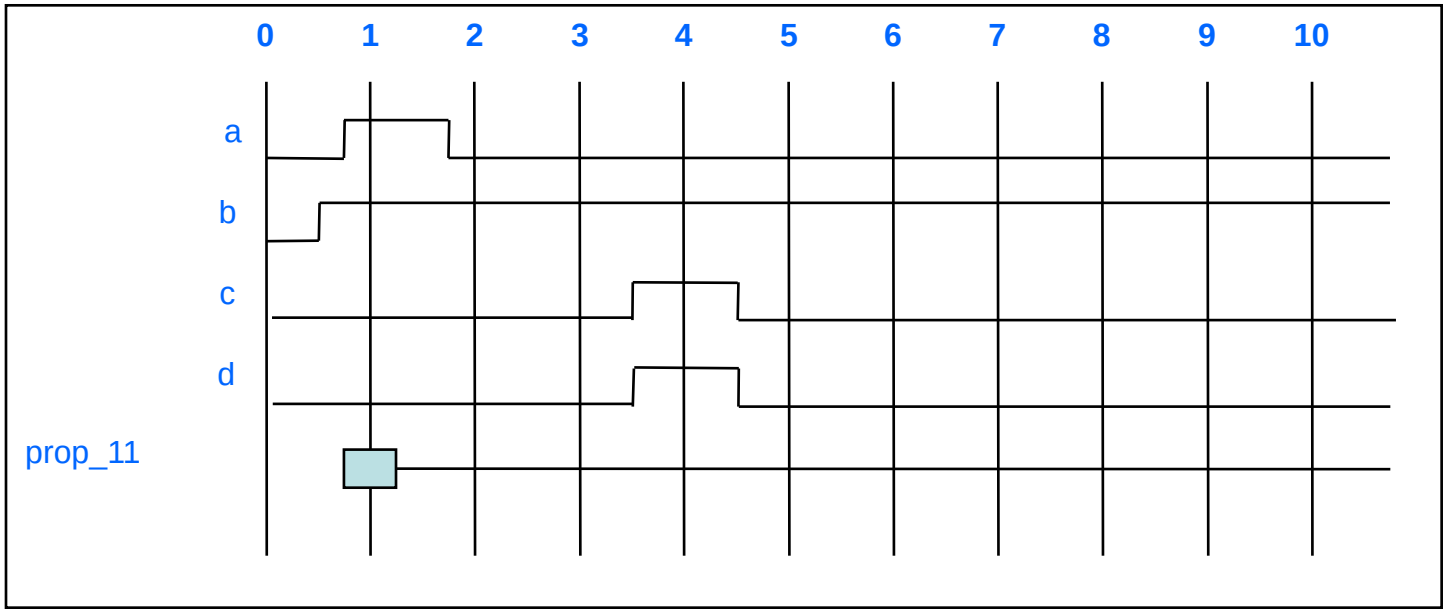
First match on antecedent

```
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);
```



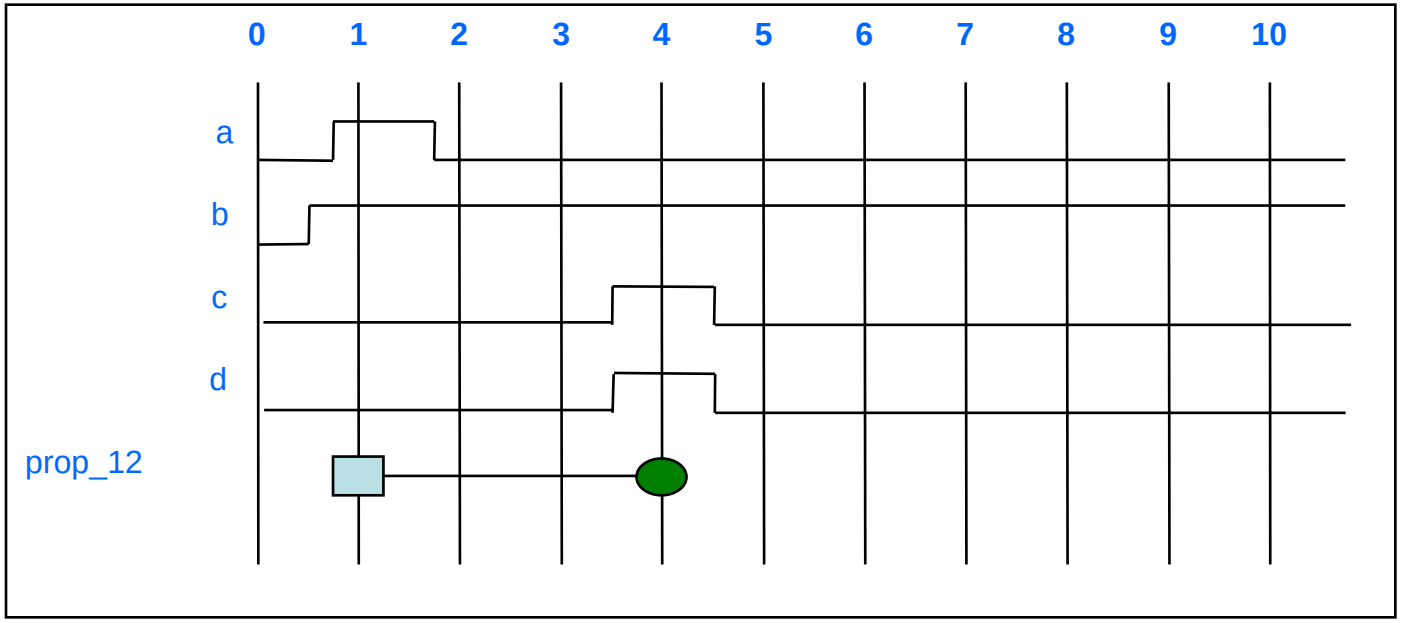
Conseqent C1 never ends





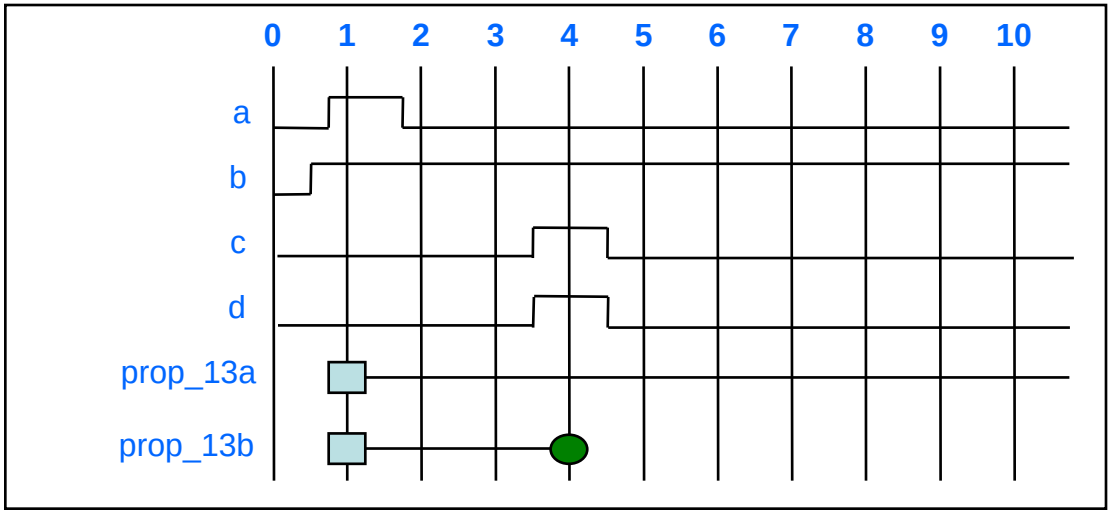
The real way to model this

```
property prop_12;
  int v_cnt;
  @(posedge clk) $rose(a) |-> first_match(b[*0:$] ##1 c) |-> d;
endproperty:prop_12

example_12: assert property (prop_12);
```



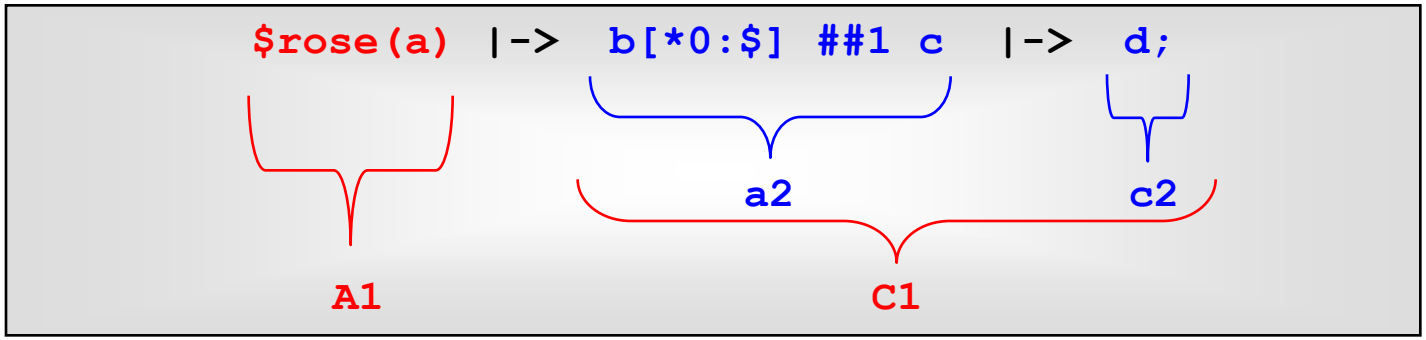
Another way to model this using fusion (##0)

```
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

example_13a: assert property (prop_13a);
example_13b: assert property (prop_13b);
```



What about vacuousity?

- either A1 and a2 failing causes a vacuous success

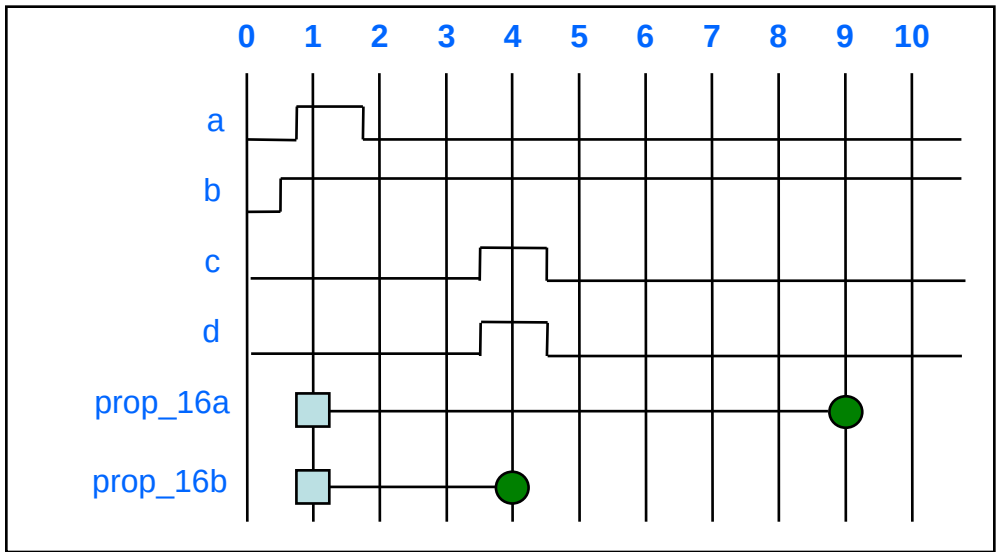


How do I model vacuousity from A1 only

```
property prop_19a;
  @(posedge clk) a |-> b |-> c ;
endproperty:prop_19a

property prop_19b;
  @(posedge clk) a |-> b;
endproperty:prop_19b

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

example_19c: assert property (prop_19c);
```

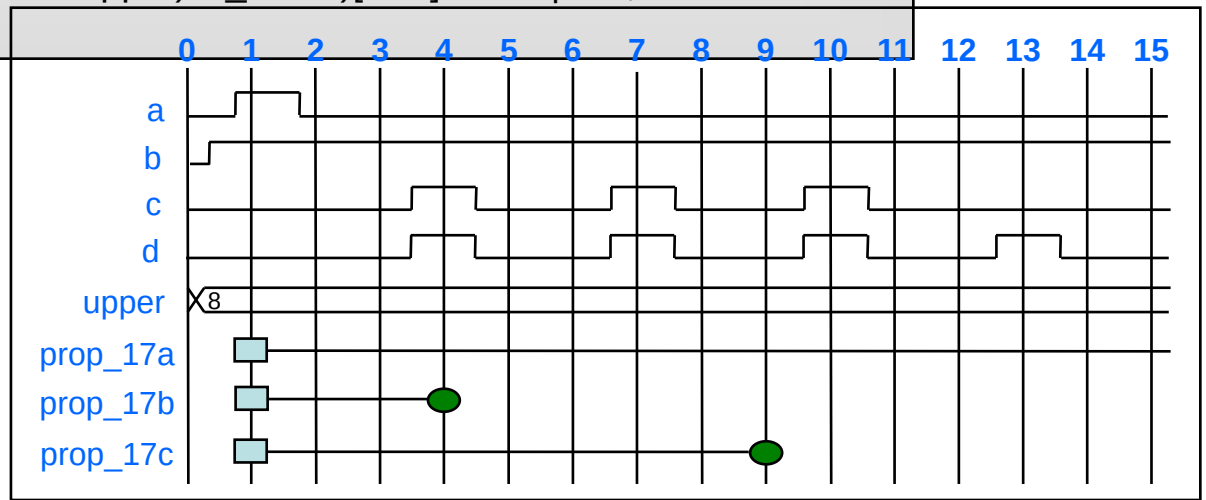
What if the upper bound was fixed and not infinity?

```
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

example_16a: assert property (prop_16a);
example_16b: assert property (prop_16b);
```



Variable upper range limit

```
property prop_17a;
  int v_cnt;
  @(posedge clk) $rose(a) |-> b[*0:$] ##1 c |-> d;
endproperty:prop_17a

property prop_17b;
  int v_cnt;
   @(posedge clk) $rose(a) |-> first_match(b[*0:$] ##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
```



Variation – only test when the variable upper limit is reached

```
module foo;
  bit a, 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;
```

Summary

- An implication with a range in the antecedent ends when
  - A passing antecedent has a failing consequent
  - The end of the range occurs
  - If the range is $
    - Will not end until all possible antecedent “passes” have tested with a passing consequent
    - Must use a first_match on the antecedent condition

```
`define TRUE 1

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);
```

Questions & Answers…

- QUESTIONS and GUESSES