The other solution is to use coverpoints, such as:

```
    covergroup a_is_x_cg;
       type_option.merge_instances = 0;
       option.per_instance = 1;
       option.get_inst_coverage = 1;
       cp_unknown: coverpoint $isunknown(a)==0;
       // ($isunknown ==0) i.e. true then no X or Z
       cp_with_countbits:  coverpoint countbits(a) ==0 ;
       cp_with_countbits_axz:  coverpoint $countbits (a, 'x, 'z) ==0; // 1800-2012
       // ($countbits (a, 'x, 'z) ==0) then no X or Z
   endgroup
```

Simulation results:

```
** Error: Assertion error.
#    Time: 30 ns Started: 30 ns  Scope: coverx.ap_aNoXZ File: coverx.sv Line: 15 Expr:
$isunknown(a)==0
```

Covergroups

Name	Coverage	Goal	% of Goal	Status	Merg
/coverx					
TYPE a_is_x_cg	100.0%	100	100.0%		0
CVP a_is_x_cg::cp_with_countbits	100.0%	100	100.0%		
INST \/coverx/t_cg	100.0%	100	100.0%		
CVP cp_with_countbits	100.0%	100	100.0%		
bin auto[1] true, No X or Z	9	1	100.0%		
bin auto[0] false, Has X or Z	1	1	100.0%		
CVP cp_unknown	100.0%	100	100.0%		
bin auto[1] true, No X or Z	9	1	100.0%		
bin auto[0] false, Has X or Z	1	1	100.0%		
CVP a_is_x_cg::cp_unknown	100.0%	100	100.0%		

Annotated coverpoint results

10.26 Uniqueness in attempted threads -- the FIFO

Requirement*:* Need to assure that each started assertion from start to completion is unique; this means that if multiple assertions are started at different cycles because of a successful antecedent, a successful consequent should not terminate all those assertions,

The following problem demonstrates the issue:
in_data is pushed into a FIFO upon a *push* control signal, data is popped out as *out_data* upon a *pop* signal. There can be multiple pushes prior to a pop.

Problematic assertion: A solution that appears plausible, but has severe issues, is the following:

```
module fifo_aa;  // /ch10/10.26/fifo_aa.sv
   bit clk, push, pop;
   int ticket, now_serving;
   bit [7:0] in_data, out_data;
   initial forever #5 clk=!clk;

   property p_data_chk_bad; //
     bit [7:0] push_data;
     @(posedge clk) (push, push_data=in_data[7:0])
               |-> ##[1:10] pop ##0 (out_data == push_data);
   endproperty
   ap_data_checker_bad: assert property(p_data_chk_bad);
```

Problem is lack of uniqueness, one pop can terminate all threads

The problem with this assertion for the fifo is uniqueness. Specifically, a `pop` can complete 2 separate threads, as shown in the simulation results for `ap_data_checker_bad` where one `pop` terminates both threads.

Figure 10.26 demonstrates the simulation result for this assertion. Note that after 2 push controls with the same value of data, both assertion threads terminate with a single pop; this is obviously not desired.

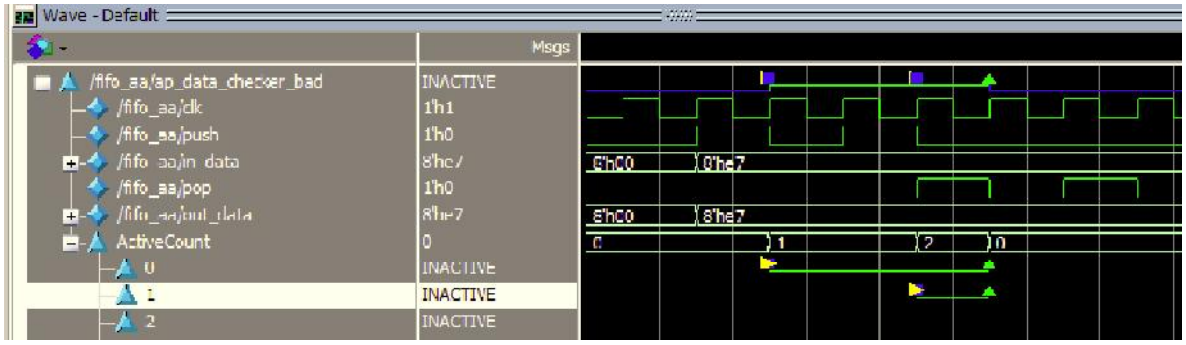


Figure 10.26 Simulation of a plausible, but incorrect assertion

A solution: What is desired for this FIFO assertion is the exclusivity or uniqueness of each attempted thread sequences, meaning that one successful consequent does not terminate all concurrent attempts waiting for that consequent.

To accomplish this, one could use concepts of a familiar model seen in hardware stores in the paint department. There, the store provides a spool of tickets, each with a number. As a customer comes in, he takes a **`ticket`**. The clerk serving the customers has a sign that reads "**NOW SERVING, TICKET #X**". The customer that has the ticket gets served, the others have to wait. When done, the number X in incremented, and the next in-line customer gets served.
The assertion code could then be written as follows:

support variable to achieve attempted thread uniquness

Function needed to increment the ticket spool in the sequence_match_item

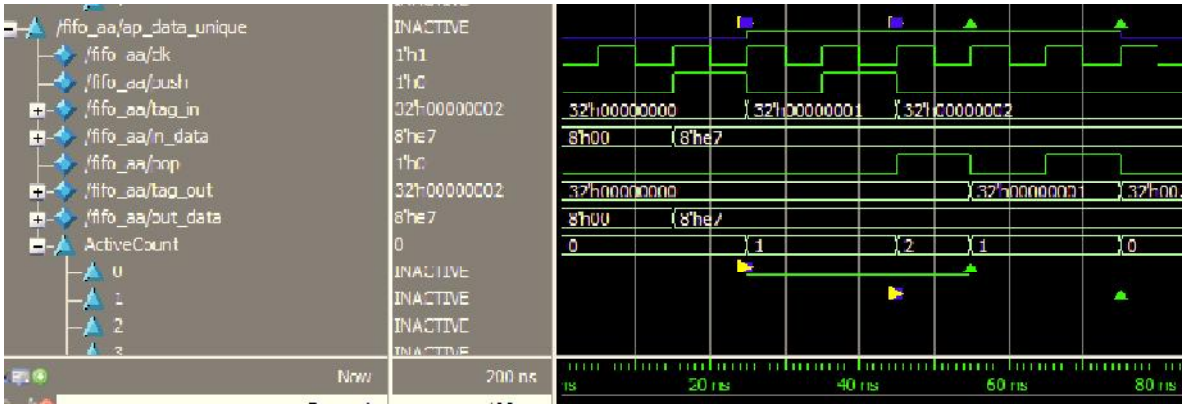
```
module fifo_aa;
       bit clk, push, pop;
       int ticket, now_serving;
       bit [7:0] in_data, out_data;
       initial forever #5 clk=!clk;

     function void inc_ticket();
       ticket = ticket + 1'b1;
     endfunction

     property p_data_unique;
       bit [7:0] push_data;
        int v_serving_ticket;
       @(posedge clk) (push, push_data=in_data[7:0],
                           v_serving_ticket=ticket, inc_ticket())
               |-> ##[1:10] pop && now_serving==v_serving_ticket
                   ##0 (out_data == push_data);
     endproperty
```

```
ap_data_unique: assert property(p_data_unique)
                now_serving =now_serving+1;
           else now_serving =now_serving+1;
```

now_serving tag incremented at conclusion of assertion for pass or fail case.



Simulationn results with code uniqueness

NOTE: If a module variable is updated through a function call in a sequence_match_item, then do not read that same module variable in the same time step. This is because all module variables are read in the Preponed region, and any update through the function call will be missed in the same cycle. For example,

```
module m_var;
    bit clk, a, b, reset_n;
    int count=0;
    default clocking @(posedge clk); endclocking
    function void upcount(int v);
       count =  v;
    endfunction

    property p_bad_style; // for demo
       int v;
       ($rose(a), upcount(2)) |-> count==2;
    endproperty
    ap_bad_style: assert property(p_bad_style)
       else $display("count= $d", $sampled(count));
```

count has the sampled value, not the updated value in the function call.

10.27 Exclusive consequent once antecedent is true

Requirement: Check that for one start-of-frame (*sof*), there should be only one end-of-frame (*eof*).

Solution: The solution depends on the interpretation of the requirements. If the requirement is uniqueness in attempted threads, then Section 10.26 addresses a methodology. If the requirement is that a *sof* must be ended with an *eof*, and until then, all other *sof* are ignored, then one could set a variable (e.g., *busy_eof*) that gets set upon a first arrival of sof (through a function in the sequence_match_item), thus rejecting other *sof*. When that thread is done it resets that variable (through a function in the action block). Another assertion states that as long as that variable is set, there should never be a *sof*.