Hi, Mike & Dave (and others!) -

There are elements of Mike's proposal that I like and this proposal raised an issue that I had forgotten; however, this proposal misses a few other important cases. Allow me to respond, expound additional scenarios and to make some alternate proposals.

Testimonial: Everyone should read Mike's paper on X-propagation problems:
www.arm.com/pdfs/Verilog_X_Bugs.pdf

Other papers worth reading include papers by Lionel Bening on 2-state simulation:

- Lionel Bening, "*Simulation of High Speed Computer Logic*," Proc. 6th Design Automation Conf., June, 1969.
- Lionel Bening, "*A Two-State Methodology for RTL Logic Simulation*," Proc. 36th Design Automation Conf., June, 1999.
- Lionel Bening, Kenneth Chaney, *Generation of Reproducible Random Initial States in RTL Simulators*, Hewlett-Packard patent US6061819, May, 2000.

NOTE: The problem of unwanted X-propagation is a 4-state simulation artifact, not a synthesis problem.

I believe what we need is the following:
(1) case (...) inside (already part of SystemVerilog)
(2) Either a pre-default in the case statement or a parallel_case equivalent case and if-modifier
(3) X-detection in testing expressions
(4) X-assignments if X-detection does indeed detect X's

For (2) above, I had proposed:
```
case (...)
  initial: //execute pre-case code then match one of the following
  caseitem_1: // code_1
  ...
  caseitem_n: // code_n
  default: // default code
endcase
```

This was largely rejected in favor of the concept of a parallel_case equivalent.

I now propose (this should also work with case (...) inside):
```
unique0 case (...)
```
This should report a run time warning if more than one case item can be matched but will not report a warning if none of the case items match (the latter is a difference between unique and unique0).

NOTE TO MIKE: unique and priority only report warnings not errors. This was changed in SV-3.1a despite objections from users. Vendors pointed out that they cannot always guarantee that the reported errors were indeed errors (for example at time-0).

For (3) & (4) above, I would like to propose five new keywords:

```
   initialx alwaysx always_combx always_latchx always_ffx
```

Any conditional or loop expression (if-expression, case-expression, while-expression, repeat-expression, do-while-expression, for-loop-expression - wait-statement may be an exception because it will wait indefinitely on an X) coded inside one of the proposed x-block (x-procedure) varieties that results in a Z or X would do the following:

(a) assign X to all variables within the scope of the conditional statement or loop that fails
(b) ignore all system tasks and functions (such as $display)
(c) and perhaps report a warning or error that the tested expression result was X or Z (??)

Mike only addressed case statements but there are other conditional and loop statements that suffer the same problems (but ARM currently does not use them due to the problems).

I think this is as easy as (i) resolve the test expression, then (ii) do XOR-reduction on the expression result, and if equal to X, do steps (a) - (c) above. This is basically a $isunknown check on the test expression.

I thought about new keyword varieties of conditional and loop commands or a different syntax arrangement to accomplish the same goal but I realized that the simplest form would be to say, "every conditional test inside this always_combx procedure should be tested for X's."

SYNTHESIS NOTE: Synthesis tools would treat these exactly the same as the non-X-procedure version (they would just be synonyms to the synthesis tool).

Now let's look at Mikes examples and re-code them per the proposed changes:

INDEX: - Problem #1
=====
Consider a simple example of a "Count Leading Zeros" circuit.

First, a case statement in Verilog 95:

```
case (sel)
  3'b000:  clz = 2'b11;
  3'b001:  clz = 2'b10;
  3'b010,
  3'b011:  clz = 2'b01;
  3'b100,
  3'b101,
  3'b110,
  3'b111:  clz = 2'b00;
  default: clz = 2'bXX; // FULL X-propagation
endcase
```

This is a correct implementation which also achieves X-propagation, i.e. any incoming X on sel will set clz to X (good chance of seeing X in simulation).

Problem is that it's verbose, even for a 3-bit selection expression. If the sel signal was 32-bits, you would need 2^32 case-items i.e. a case statement spanning 429 million lines!

A concise form can be written in casez:

```
casez (sel) // Z in sel is a wildcard
  3'b000:  clz = 2'b11;
  3'b001:  clz = 2'b10;
  3'b01?:  clz = 2'b01;
  3'b1??:  clz = 2'b00;
  default: clz = 2'bXX; // PARTIAL X-propagation e.g. 3'b01X
                        // is terminated (clz=2'b01)
endcase
```

This is concise, just 32 lines rather than 429 million, but has two issues:

a) X on sel is not propagated in all cases
b) an incoming Z in sel is a wildcard

Issue (b) can be addressed in SystemVerilog by using "case inside", because it only allows wildcards on the RHS

```
case (sel) inside
  3'b000:  clz = 2'b11;
  3'b001:  clz = 2'b10;
  3'b01?:  clz = 2'b01;
  3'b1??:  clz = 2'b00;
  default: clz = 2'bXX; // PARTIAL X-propagation e.g. 3'b01X
                        // is terminated (clz=2'b01)
endcase
```

Problem (a) is still real though, as X-propagation is incomplete. A workaround would be to add an OVL x-checking assertion on the sel[2:0] signal.

ASIDE: by using "unique case" in SystemVerilog you can avoid using the "default" and cause a runtime error when X is detected. However, we still have problem (a) because some X's would match earlier "?" wildcards.

NEW - Solution #1

With the proposed solution we can fix both problems with:

```
always_combx
  case (sel) inside
    3'b000:  clz = 2'b11;
    3'b001:  clz = 2'b10;
    3'b01?:  clz = 2'b01; // Matches 3'b010, 3'b011
    3'b1??:  clz = 2'b00; // Matches 3'b100, 3'b101, 3'b110, 3'b111
  endcase
```

If sel has any X-bits or Z-bits, clz will be set to 2'bxx. No default necessary. Even more concise than Mike's proposal

INDEX: - Problem #2
=====
Consider a decoder circuit where some values set an output high but most set it low.

```
     reg        match;
     reg [15:0] instr;

     case (instr)
       16'h014F: match = 1'b1;
       16'h152E: match = 1'b1;
       <...>
       default:  match = 1'b0; // PROBLEM: X-termination
     endcase
```

NEW   - Solution #2

```
   always_combx begin
     match = '0;
     unique0 case (instr) inside
       16'h014F: match = '1;
       16'h152E: match = '1;
       <...>
     endcase
   end
```

Pre-assignment match='0 takes care of the binary default case. The unique0 keyword says that instr will match only one of the listed case items or none of the listed case items. No default of any type is required because if instr has any X's or Z's, the last assignment to match will be X.

The reason that this style is so important (a case that Mike did not expound) includes most Finite State Machine (FSM) Designs. Consider an FSM with 10 different outputs. One way to code this FSM is to make all 10 assignments for each state of the FSM (very verbose, inefficient, confusing and error-prone) or to make default assignments to the outputs just before entering a case-statement, then make just the required updates to the outputs for the required states (more concise, efficient, easy to follow and less error prone). With the new proposals:

```
   always_combx begin
     next = 'x;  // default next assignment
     out1 = ...  // default output assignments
     out2 = ...
     ...
     out10 = ...
     unique case (state)
       IDLE:                    next = STATE_1;
       STATE_1: begin // only assign modified outputs
                  out2 = ...
                  out7 = ...
                  if (in1)      next = STATE_2;
                  else          next = STATE_1;
                end
```

```
      STATE_2: begin // only assign modified outputs
                 out1 = ...
                 out3 = ...
                 out4 = ...
                 if (in2)      next = STATE_3;
                 else if (in3) next = STATE_4;
                 else          next = STATE_2;
               end
      ...
    endcase
  end
```

In this example, if state resolves to X, all assigned outputs go to X and next is assigned to X. If while in state_2, in2=X or Z, next is set to X (but the output assignments complete correctly for STATE_2).

By using the always_combx style, I don't have to use modified versions of cases statements or multiple modified versions of if-statements. Very user-friendly for RTL designers.

MORE EXAMPLES:

Synthesizable flip-flop:

```
  always_ffx @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d;
```

If we had used an always_ff procedure and if rst_n had not been defined or was X, The FF code would assign q <= d, which is not necessarily correct. Using always_ffx, q would be assigned the value of X to help simulations to fail when they should. Synthesis note - no different than always_ff.

Synthesizable latch:

The same type of problem can be avoided when coding a latch:

```
  always_latchx
    if (!rst_n)  q <= '0;
    else if (en) q <= d;
```

This code will trap X's on either rst_n or en.

More examples could be provided.

Regards - Cliff