

Contributed Article

 [Print](#)  [Email](#)  [Bookmark](#)

Formal verification enables safe X handling

By Rajeev Ranjan, Yann Antonioli, Alan Hunter, and Oleg Petlin

12/16/08

Introduction

The use of unknown values (X states) can provide significant benefits in RTL verification, and can allow better logic optimization for synthesis. But uncontrolled or unexpected X propagation in simulation can also mask bugs, potentially leading to failures in silicon. Simulation alone cannot adequately simulate and observe all X states. Formal technology can provide a much more complete solution that allows engineers to derive the benefits of X states without the associated risks.

Simulation cannot easily control Xs and ensure that they propagate to observable points, or that they do not propagate to design points (primary outputs or internal nets) where real Boolean values are expected. Depending on coding style, simulation can be overly optimistic, generating real values when signal values are really unknown – or pessimistic, producing Xs when signal values could be known. The gate-level interpretation of “X” can result in a synthesized design that doesn’t behave as expected.

Formal verification, unlike simulation, can evaluate the design for both possible values of X (0 and 1) at each clock cycle. It exhaustively traverses all the paths in the design, and can find corner-case conditions where Xs propagate to observable points. As such, it offers much better prospects for controlling X propagation and ensuring safe use of Xs. What’s needed now is an approach that eases the burden of writing specifications, helps users easily debug problems, has high capacity, and uses industry-standard specifications that support portability.

This paper will draw upon experience from ARM and AMD to show how X states are used in synthesis and verification. It will show how simulation and formal verification handle X propagation today, and outline some of the requirements for an ideal solution based on formal technology.

What X states are and how they’re used

Different semantics are associated with X for different parts of the design flow. X is interpreted as “don’t care” in synthesis and “unknown” in simulation. In RTL design, X tells the synthesis tool that it doesn’t matter whether a 0 or 1 is assigned during logic optimization. In verification, X tells the simulator that a signal value is unknown.

There are three basic sources of X. The first of these is an explicit assignment of X in logic optimization, which allows the synthesis tool to choose either a 0 or 1 to improve logic minimization. This can have unintended consequences during verification, such as the X-optimism or X-pessimism described in the next section.

In the second case, X is explicitly assigned for verification purposes. For example, you could have a case statement in which four cases are enumerated – 00, 01, 10, 11 – and then have a default to which you assign an X. These Xs appear in RTL code as 1’bx. They are added to the RTL to uncover unwanted conditions. For example, if a selector signal that should be one-hot is not because of incorrect or uninitialized logic, the case statement goes to the X default and is propagated towards an observable output (see Figure 1).

```

case(busSelect):
  3'b001: data = dataA;
  3'b010: data = dataB;
  3'b100: data = dataC;
  default: data = 10'bx;
endcase
    
```

(a)

```

case(busSelect):
  2'b00: data = ata;
  2'b01: data = dataB;
  2'b10: data = dataC;
  2'b11: data = dataD;
  default: data = 10'bx;
endcase
    
```

(b)

Figure 1: An example of an explicit X assignment (a) to detect non-one-hot values of "busSelect" and (b) to detect unknown values on "busSelect."

There are many ways to use explicit X states in verification. To cite one example, you could assign the first 8 bits of a bus to known values, and the remaining bits to X. This could make sense in a situation where you know that only a subset of the bus bits will be utilized.

In the third case, uninitialized registers automatically result in unknown values in simulation.

Intentionally using uninitialized registers can be regarded as an implicit X assignment. It is often done to avoid wasting area with initialization. Datapath busses are expensive to initialize, and by tracking X states, you can potentially ensure that a bus is not accessed before an event occurs that makes the values on the bus valid. In the code example shown below, the design must ensure that the appropriate address in the memory gets proper data *before* an attempt is made to read that memory location.

```
always @(posedge clk)
    if (wrEn) mem[wrAddr] = dataIn;

assign dataOut = rdEn & valid ? mem[rdAddr] : 10'bX;
```

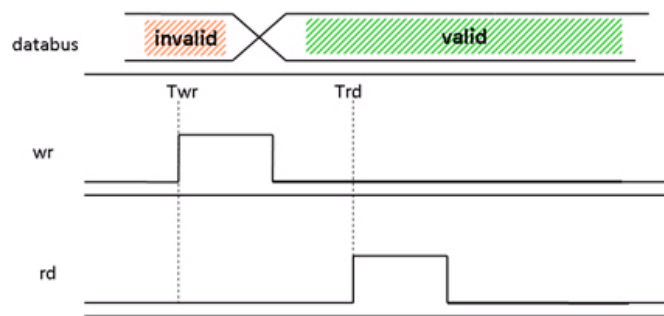


Figure 2: An example of uninitialized registers

Finally, unintentional X states can occur if a net is not driven during simulation. If a net or register has not been written into, and it is accessed, it will produce an X.

Engineers at ARM generally avoid using explicit X states in synthesis and verification. That's because X propagation can cause inconsistencies between simulation behavior and silicon implementation. However, they do implicitly assign Xs by using uninitialized datapath registers, because setting all the registers would waste a great deal of power and area. They don't initialize full arrays of memories, and that's another source of Xs coming into the design.

ARM uses assertions that check for undesired Xs. For example, the company's coding guidelines say that there should be an X default for every case statement, and assertions check to see whether the default case is ever hit. Assertions also check the requirement that AMBA bus signals should never be Xs.

At AMD, engineers use X states in synthesis so that optimization can produce the smallest possible circuit. This helps save area and power, and improves routability. But it can potentially result in ambiguities that cause discrepancies between RTL and gate-level simulation.

Because of the difficulty of debugging X propagation, AMD engineers generally avoid generating Xs from testbenches. AMD also generally discourages the use of uninitialized registers, but does allow a certain percentage to be uninitialized in order to conserve real estate. Designers are encouraged to use assertions to ensure that uninitialized registers don't flood the design with unwanted Xs.

Managing X propagation – where simulation falls short

It should be noted that X propagation is not necessarily a bad thing. If it's intended, it can be a powerful tool for catching bugs in a design. Ideally, if an X occurs at an observable point, you'll be able to step backwards and identify where the X assignments occurred. You can then modify the design to remove the source of the uncertainty.

Unintended X propagation, however, is a problem. If an X propagates to a primary output, and you go ahead with synthesis and tapeout, the chip might have an unknown value appearing at the output, and neighboring logic won't be able to process it correctly. As a result, functional behavior at the gate level or in silicon may differ from what you intended in the RTL code, leading to debugging problems or even failed silicon.

When X states occur, simulation cannot check all possible combinations of 0 and 1 corresponding to the X states. In addition, simulators can't run enough scenarios to catch all X propagation issues. Consequently, they will miss corner cases that cause unintended X propagation. Further, if code is not written properly, Xs can result in both optimistic and pessimistic results in simulation.

X-optimism occurs when the interpretation of X takes just one if/case branch when many should be

considered¹. It thus produces a binary value of 0 or 1 when the actual state is unknown. You might have monitors set up to check for Xs at points where they should not occur, but due to X-optimism, the Xs are blocked. As a result, the behavior of the netlist and/or silicon implementation will differ from the RTL.

An example of X-optimism is as follows:

```
always @ (posedge clk)
  if (reset)
    Count <= 3'b0;
  else if (CountEn)
    Count <= NxtCount;
```

Figure 3: X optimism

Suppose a reachable don't care X is assigned to CountEn. The else if branch will only execute if CountEn is 1'b1, so no update occurs if CountEn is X, and the count will keep its previous value. Effectively, the simulator is treating the X as a 0 when in reality it could be either a 0 or 1.

X-pessimism occurs when results lead to more X propagation than is really necessary. It thus generates Xs where there are actually binary values. X-pessimism is demonstrated in the following example:

sel	statusA	statusB	status
1	0/1/X	—	statusA
0	—	0/1/X	statusB
X	—	—	X

Figure 4: X pessimism

The simulation interpretation of the assignment is:

```
assign status = (sel & statusA) | (~sel & statusB);
```

Suppose statusA and statusB have equal values, say 1'b1 or 1'b0. In simulation, when sel is set to X, the value of status is always evaluated as 1'bX. In reality, status should be equal to statusA and statusB regardless of the value of the select line.

[Next Page](#)

Company Contact



Corporate Headquarters
100 View Street, Suite 101
Mountain View, CA, 94041 USA
+1.650.966.0200 Phone
+1.650.625.9840 Fax

Learn more

- [Jasper Newsletter](#)
- [Whitepaper Request](#)
- [The Dangers of Living with an X \(Mike Turpin, ARM Ltd.\)](#)

In case you are not able to click one of the hyperlinks above please make sure to be [registered](#) and [logged-in](#).

Request for Information

Your message:

You must be [registered](#) and [logged-in](#) to submit an rfi.

Add Comment - please [log-in](#) to comment

SCDsource newsletter subscribers may post a comment - [Register for free!](#)

[Back to Home Page](#)