

Using Formal to Analyze Non-Determinism in Design Reset Schemes

By Chris Browy and Kai-hui Chang

Reset schemes can be difficult to verify in logic simulation because of the non-determinism caused by unknowns (Xs) in the registers and their inaccurate handling in logic simulation which can mask bugs and potentially lead to failures in silicon. Here, we describe a precise formal approach to X-verification of partial and full reset sequences. Insight, from Avery, addresses two kinds of X issues:

- 1) X propagations are accurately and fully disclosed so that they can be fixed in RTL
- 2) Reset simulation uncontrollability, as a result of physical synthesis optimization of reset logic, is analyzed and fixed so that gate-level logic simulations of reset works properly.

Limitations of Logic Simulation of Xs

Logic simulation is the most widely used verification technology and may be able to find some X-problems in a design. However, X-handling in logic simulation is often inaccurate due to X-optimism where 0/1 values are propagated instead of a real unknown, and X-pessimism where Xs are propagated even though the 0/1 value can be known.

Take the simple code shown in Figure 1a as an example, if “a” is 1’bx, “out” can be either “b” or “c”. However, due to X-optimism in logic simulation algorithms, only one branch is considered and “out” will be equal to “c”. Figure 1b shows another example where signal “out” should not be affected by “a” but is assigned X erroneously due to X-pessimism. Worse yet is the case in Figure 1c. Here an inequality comparison of 4 bit registers, “a” and “b”, wrongly sets “out” to 0.

(a) X-optimism a = 1’bx; if (a) out = b; else out = c; result: out = c;	(b) X-pessimism a = 1’bx; b = 1’b1; c = 1’b1; out = (a & b) (~a & c); result: out = 1’bx;	(c) X-wrong a = \$’b1xxx; b=4’b0100; if (a > b) out = 1; else out = 0;
---	--	---

Figure 1: A simple example to show X-optimism and X-pessimism problems in logic simulation.

Xs originate in designs for several reasons:

- Inputs, registers, or memory that have not been initialized or reset
- Bus clash on tri wire nets
- X corruption on power down cycle in low power designs
- X assignments intended for better optimization by synthesis tools
- X assignments to catch unexpected conditions in verification

Common examples of X assignments for unexpected conditions in verification are shown in Figure 2. In Figure 2a, a 3-bit control variable has only 3 valid encodings. In this case if the design generates an invalid encoding then the output will be assigned to X. In Figure 2b, a status bit determines if the memory output is valid to drive. If not the X assignment helps the designer isolate a potential bug.

(a) X assignment for control problem

```
case(control)
  3'b000: data_out = source_1;
  3'b010: data_out = source_2;
  3'b101: data_out = source_3;
  default: data_out = 32'b x
endcase
```

(b) X assignment for invalid data

```
if (status)
  data_out = mem[addr];
else
  data_out = 32'bx;
```

Figure 2: X assignment for verification purposes

The problem with these X assignments is that while they do reflect the presence of non-determinism in the design which can occur for valid reasons, the effects of the X propagations cannot be evaluated properly in logic simulation to fully determine if there are unwanted side effects. To do so would require modeling the behavior of X for all conditional expressions. For example, in Figure 3, if the data_out register were to be conditionally loaded based on a load_enable then to model the non-determinism would require the following extra checking:

```
always @(posedge clk)
  if (load_enable === 1'x)
    data_out_reg = 32'bx;
  else if (load_enable === 1'b0)
    data_out_reg = data_out_reg;
  else
    data_out_reg = data_out;
```

Figure 3 RTL explicitly models non-determinism

However while this modeling technique would potentially propagate the non-determinism more accurately in the RTL it comes with a heavy burden for the designer. It is also very difficult to code n-bit reg behavior.

Another example of inaccurate X propagation is when a memory is read prior to being written with valid data. The resulting X on the output of the memory may not properly propagate due to X-optimism.

Some designers take a completely different approach and try to remove non-determinism all together. They do this one of several ways:

- Use simulation runtime options to randomly initialize them to either 0 or 1
- Use the SystemVerilog bit data type which only supports 2 states (0/1)
- Flag all invalid encodings at the source and use immediate assertions to notify the designer

This limits the verification by removing non-determinism all together however and just “pushes the can down the road” until there are RTL vs gate-level simulation mismatches.

On the other hand, formal analysis is not burdened by these simulation limitations. Formal analysis can follow all execution paths due to the non-determinism of Xs and warn designers when divergent design

behaviors result. The following sections will consider the reset sequence as one prime situation when Xs are prevalent and the designer needs to make sure that the design operates properly after reset.

Reset Methodologies

Design reset methodology follows two schools of thought:

1. Full reset – use logic or software routines to initialize all physical registers, or
2. Partial reset – use logic or software routines to initialize only a subset of the physical registers that are considered essential.

Either approach has to cope with several design challenges which can be very costly because:

- Inserting the reset tree requires techniques similar to clock tree insertion and requires balancing latency and loading. Due to the nature of layout tasks, this process may require ripping up and re-placing and re-routing regions of the design.
- Logic insertion to ensure that state machines are not out-of-sync immediately after reset is de-asserted.
- Timing constraints are typically relaxed for the reset tree; therefore the internal power-on/reset (POR) generator typically requires an additional programmable reset pulse width stretcher to cover tree latency to ensure all registers are properly reset.

Reset	PRO	CON
FULL	Predictable bring-up state	Higher routing resources Increased potential for timing closure issues due to routing congestion and limited cell sites for reset buffers
PARTIAL	Lower routing resources Lower potential for timing closure issues	Design complexity due to manual register selection process RTL vs gate-level simulation mismatches Need to manually analyze design for possible unexpected design operation due to non-determinism in bring-up state

Table 1: Pros and cons of full and partial reset

The choice of applying partial or full methodology is based on design requirements, vendor handoff requirements, or on avoiding past situations that posed problematic. One such problem is functional non-determinism during design bring-up, because improper upfront design and verification methods

result in the bugs only being found in lab prototypes. Resolving any reset-related issues at the physical design stage can take weeks or even months to be resolve.

Overall the design and verification tradeoffs associated with the choice of full or partial reset are summarized in Table 1, and deal with physical design and functional verification.

From the standpoint of optimizing a design, partial reset would always be the favored approach if it were not fraught with such undesirable outcomes when poorly executed. Typical design benefits include:

- 1) reduce die size of the implementation
- 2) reduce cost
- 3) reduce number of objects in design database to be handled by physical design tools
- 4) make overall design timing easier
- 5) permit routing closure
- 6) reduce runtimes of physical design tools

Deciding what to reset and software initialize to ensure 100 percent reliable bring-up operation after POR or hot reset is a manually-intensive and iterative process that involves detailed analysis to remove any adverse operational non-determinism due to Xs in registers.

Reset sequences can often involve thousands of clock cycles while the hardware resets, and software drivers or firmware initialize the design (see Figure 4). The ROI for this effort is based on trading-off the engineering cost to perform manual selection and verification versus the mitigation of timing closure and routing problems.

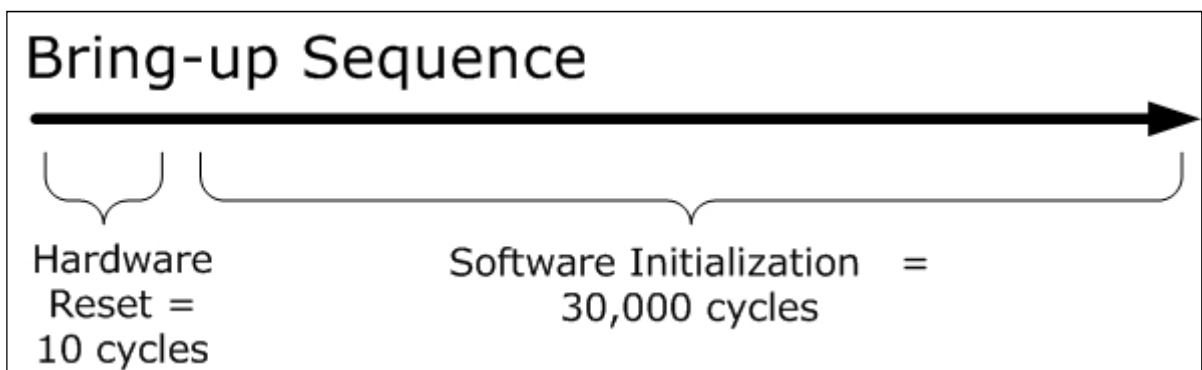


Figure 4: Typical design reset and software initialization sequence

Some rules of thumb are used to select registers to apply hardware or software reset and are summarized in table 2.

Registers that need reset	FSM registers, key control registers, registered inout ports, and memories (usually via hardware reset) Software visible registers and memories (usually via software reset)
Registers that do not need reset	Datapath and staging registers Memory buffers Non-architectural registers

Table 2: Registers that require reset and those that do not

The partial design reset and analysis process comes with numerous challenges. First you would prefer to do this analysis at the RTL. However, logic simulation is known to provide inaccurate simulation of unknowns (Xs), and hence makes the verification unreliable as illustrated in the earlier examples. Here, the issues with X-pessimism and X-optimism result in possible imprecise simulation which may not catch a reset issue. To overcome this limitation, the analysis usually has to be repeated during gate-level simulation to look for RTL vs gate-level simulation mismatches because X propagations are more accurate at the gate-level. This requires running a rich enough set of gate-level simulations to activate all functional blocks of the design. Unfortunately, this approach has the effect of increasing gate-level simulation time, which engineers are trying to reduce or eliminate. Furthermore, it does not ensure 100 percent verification.

One final area of concern is generated resets, such as reset signals created by a state machine. These types of reset create unintended reset events that might not be observed without 100 percent verification.

Formal X-verification: A new approach

Formal analysis lends itself nicely to analyzing a design for non-determinism through an application called X-verification. For designs that employ partial reset and initialization there are a lot of Xs in the post-reset state, some of which could result in unexpected behavior. X-verification can reliably and exhaustively analyze all execution paths of the design to see whether these Xs can propagate and affect design behavior. Because formal analysis treats Xs as 0 or 1 simultaneously, X propagation can be performed accurately and completely at the RT level.

X-verification analyzes the complete reset and software (SW) initialization sequence and generates a counter-example and testbench to enable the designer to replay and observe the X-propagation situations in logic simulation. A typical flow is as follows:

1. Logic simulation to seed state (optional)
2. Symbolic simulation through reset and SW initialization cycles. Use interval checkpointing to analyze long sequences

3. Generate X backtrace report
4. Generate counter-examples and testbench for debug

The example in Figure 5a illustrates X-verification. The design has registers d, e and g left uninitialized, which results in non-deterministic operation prior to assigning actual 0's or 1's to the data inputs. RTL logic simulation would not indicate any non-determinism in design bring-up.

Symbolic simulation is carried out for the reset sequence period of 20 time units. Initially the RESET_ENABLE define is not used to illustrate the effect of leaving "e" uninitialized. Only one interval checkpoint at the end of the reset period is used. The result (see Figure 5b) is that register "g" is shown to be X at time 20 because of non-determinism of register "e" and "g" at time 5. The result indicates that when "e=1" then "g" is "X". However if "e=0", then "g" is "0". X-verification produces a list of registers (see Figure 5c) to add reset which fixes the problems. If reset is added to "e", then the non-determinism on "g" is eliminated. Running X-verification a second time with RESET_ENABLE confirms this as shown in Figure 5d.

X-Verification can also generate a logic simulation testbench to replay both cases of non-determinism of "e" so that the designer can fully evaluate the X issues. The testbench code in Figure 5e is generated and added to the design example above to illustrate the non-determinism of "g" in logic simulation.

```
1 module test;
2   reg e, g, d;
3   reg clk;
4   reg reset;
5
6 initial begin
7   clk = 0;
8   reset = 0;
9   end
10
11 always #5 clk = ~clk;
12
13 always @(posedge clk) begin
14   `ifdef RESET_ENABLED
15     if (reset) begin
16       e <= 0;
17     end
18   else
19   `endif
20   begin
21     if (e)
22       d <= 0;
23     else
24       g <= 0;
25     end
26   end
27
28 initial begin
29   reset = 1;
30   #10 reset = 0;
31   end
32 initial begin
33   // enable X-verification for 20 units
34   $symbolic_reset(20);
35   #25;
36   $finish;
37   end
38 endmodule
```

Figure 5a: Design example for X-verification

```
M2508: SAT ran successfully with the patterns to satisfy output required at 20. [Simulat]
Symbolic X values propagate to test.g due to the following variables and values:
At time 5, test.e: value1= 1'b0, value2= 1'b1
At time 5, test.g: value1= 1'b0, value2= 1'b1
```

Figure 5b: X-verification result

```
test.e
test.g
```

Figure 5c: X-verification repair list

```
> vck.exe test.v +define+TRACE2 +define+X_test_g__20 +define+REPLAY
Compiling source file "test.v"
Including source file "xver.v"
Continuing compilation of source file "test.v"
Highest level modules:
test
test.g = 1 @20
L48 "test.v": $finish at simulation time 25
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in checking
End of Verilog Checker 1.4.0.0710, start=Sep 17, 2009 21:09:28, end=Sep 17, 2009 21:09:28
```

Figure 5d: Logic simulation log file for one valuation of "e" resulting in "g=1" at time 20

```
1 `ifdef X_test_g__20
2 initial
3   #20 $display("test.g = %b", test.g );
4 initial
5 begin
6   #5;
7   test.e =
8   `ifdef TRACE1
9     1'b0;
10  `endif
11  `ifdef TRACE2
12    1'b1;
13  `endif
14 end
15 initial
16 begin
17   #5;
18   test.g =
19   `ifdef TRACE1
20     1'b0;
21  `endif
22  `ifdef TRACE2
23    1'b1;
24  `endif
25 end
26 `endif
```

Figure 5e: Replay testbench generated to illustrate non-determinism of “g” in logic simulation

How to verify

Generally, design stimulus can be generated using either a testbench or a VCD file. A testbench is extremely useful when it is available and has the advantage that X-verification will verify Xs under all possible inputs that the testbench can generate. For example, if variable 'A' is \$random, X-verification will check whether X propagates to observation points no matter what value “A” is. This is the most powerful way to use X-verification.

Sometimes testbenches are not available but the reset sequence is fixed. In this case, we can use VCD files as input. In this method, we replace the Xs at the primary inputs with symbols. The Xs in design registers are also treated as symbols. The advantage of this approach is that no testbench is required, and setup is very easy. However, X-verification can find Xs only under the given trace.

What to verify

One problem in X-verification is to determine what to check because Xs may still exist after reset. To solve the problem, X-verification can use assertions or waveforms.

(1) Assertions: one can write SystemVerilog assertions with \$isunknown. In the assertion, implications (\rightarrow , \Rightarrow) can be used to specify preconditions, and then \$isunknown can be used to catch Xs. If these assertions are available, then X-verification can automatically utilize them to find all the Xs that are interesting to the designers. This approach is suitable when the designer knows the architectural register set of interest.

(2) Waveforms: most of the time, assertions are not available in the design, and designers may not know what to check in the design. An all-inclusive methodology is used here that checks all the non-X values in the waveforms so that designers can believe their waveforms. The idea is that if a register already has X in logic simulation, the designer probably knows that already so this is not a problem. On the other hand, if logic simulation shows non-X but formal X analysis can prove that it is X, then the designer probably doesn't know about this, and there may be a problem. The user can manually specify the registers to check. Alternatively, the user can also use a VCD file, and formal X analysis will automatically check all the registers that do not have Xs in the waveform.

When to check and scalability improvement

Ideally, designers should check Xs every cycle to make sure there is no unexpected X. However, most often, designers check Xs only at the end of the reset sequence. This is acceptable if symbolic simulation can be performed from the beginning to the end of the reset sequence. However, if scalability becomes an issue, one can partition the design and run X-verification on each partition sequence. Such partitioning is conservative and may introduce false positives but will not miss bugs. The flow in Figure 6 illustrates how large designs are auto-partitioned and analyzed block by block. The block-level stimulus is extracted from the chip-level VCD file.

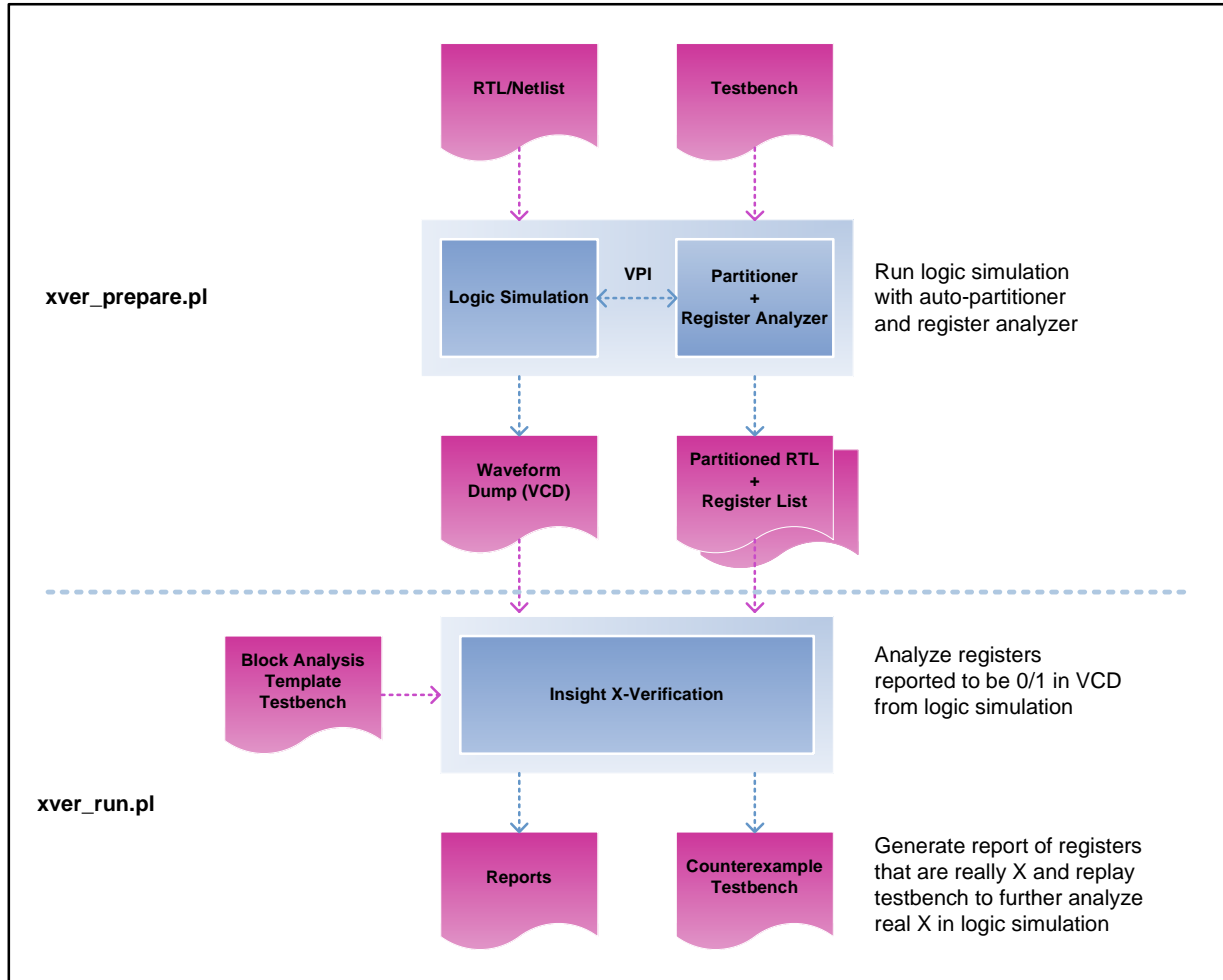


Figure 6: X-verification flow incorporating auto-partitioning of design

Real results

X-verification case studies have proven to be valuable for performing verification of the reset and initialization sequence. The solution has been shown to be scalable to multi-million gate blocks with reset bring-up sequences of 40,000 clock cycles. X-verification found numerous cases where logic simulation masked real Xs (X-optimism) that caused real hardware bring-up problems.

Reset Controllability

Another type of X problem arises at the physical design stage. Here in order to design lower-power and higher-performance circuits, aggressive physical synthesis optimizations are being used more frequently. For example take the original synthesized netlist in Figure 7a. One physical synthesis optimization is to move the reset signal of a register to its input cone, as shown in Figure 7b. One advantage of this approach is that a smaller register can be used because the reset signal is no longer required. In addition, routing may also become easier since the reset signal does not need to reach the register directly. However, this optimization causes problems in gate-level simulation due to X-pessimism. More

specifically, Xs are often used to represent unknown values in registers before they are initialized. As Figure 7b shows, “1 AND X” becomes X and erroneously makes the output of the NOR gate X instead of 0. This X will propagate to other parts of the design and corrupt the whole gate-level simulation. One way to solve this problem is to force the registers to 0 by assuming that all registers will be properly initialized after the reset period. This approach, however, may be dangerous since physical synthesis tools may have bugs. In addition, some registers may not have the reset signal at all and forcing them to 0 may mask real bugs. As a result, it is imperative to find a way to solve this problem so that gate-level simulation can be performed accurately.

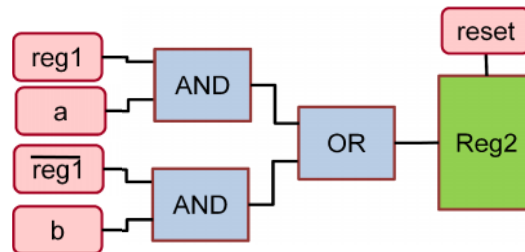


Figure 7a Original netlist

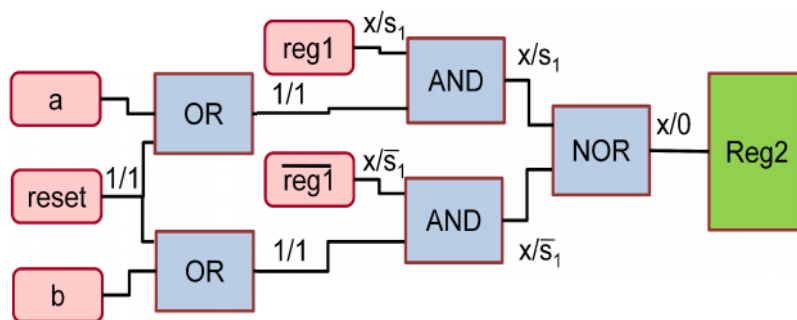


Figure 7b Netlist after optimization where the reset signal has been moved to Reg2’s input cone.

When reset 1 is 1, input to Reg2 should be 0 even though reg1 is X. However, due to X-pessimism in logic simulation, Reg2 will be X instead of 0. On the other hand, formal analysis uses a symbol (s_1) to represent the value of reg1 and can prove that the register is, in fact, a 0. (Simulated values for each wire are shown using logic value/symbolic value).

What to verify

Similar to the X-verification discussed in the last section, X-verification can be performed on the gate-level netlist after physical optimization. By proving that potential Xs are really constant 0/1 values, the gate-level logic simulation can be “fixed” through the use of a combination of force and release statements that correctly reflect the reset dominance in the fanin cone of the register. The flow in Figure 8 illustrates how large designs are auto-partitioned and analyzed block by block. The block-level stimulus is extracted from the chip-level VCD file.

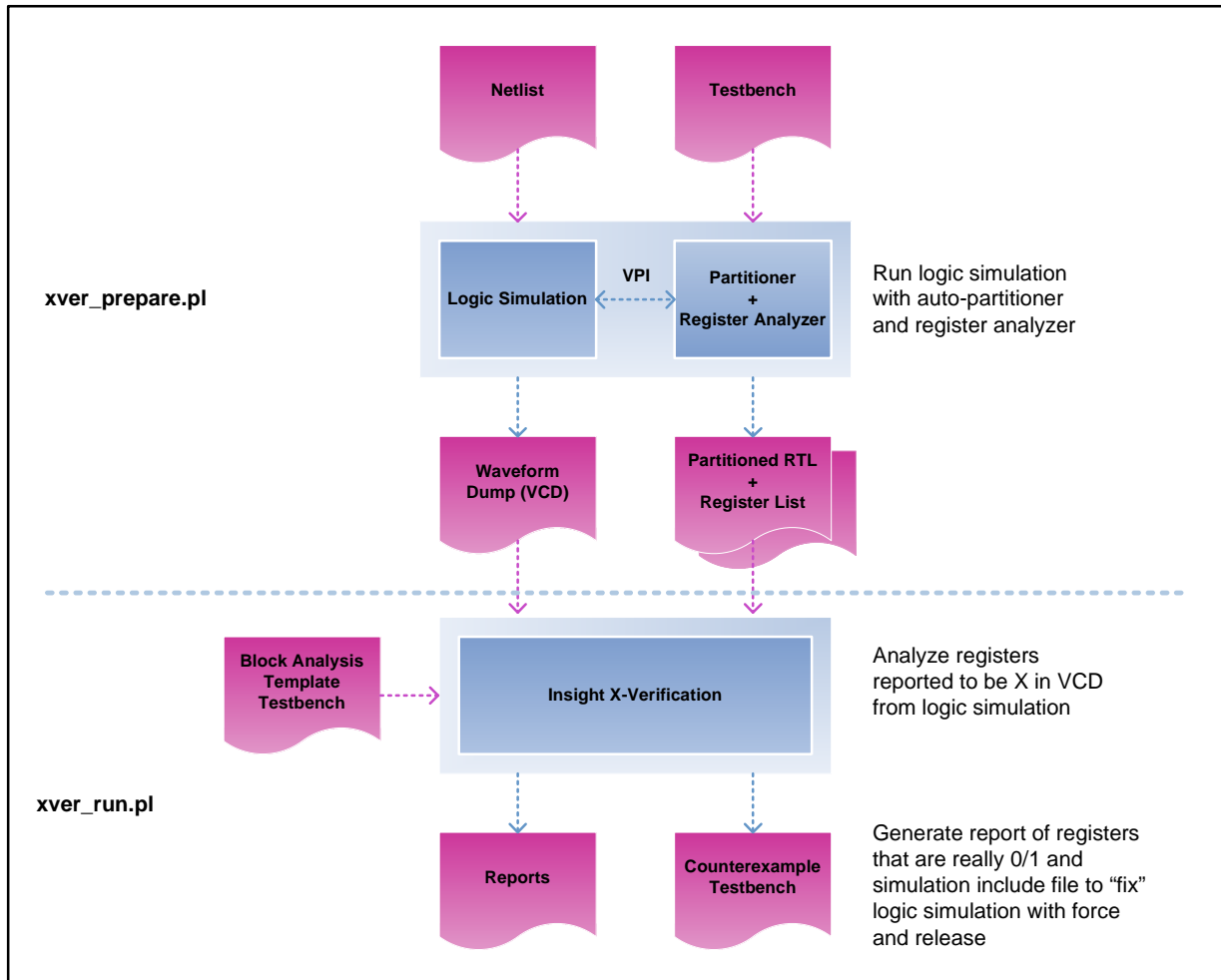


Figure 8 Reset Controllability Flow

Example Results

X-verification results indicate which registers appear to be X in logic simulation however through formal analysis can be proven to be really 0/1. For example the following example report indicates that a design register, p1_0, is 0 at time 20 and does not evaluate to X even though the simulation monitor indicates the simulation value is X.

Symbolic values do not propagate to top.i_dut.p1_0 at time 20, constant value= 4'b0000

Figure 9: X-verification result

Based on this analysis one solution to fix the logic simulation is to augment the testbench to force the p1_0 register to 0 during reset.

```
initial begin
  @(negedge reset_n)
  force top.i_dut.i_D.p1_0 = 0;
  @(posedge reset_n)
  release top.i_dut.i_D.p1_0;
end
```

Figure 10: Generated include file “fixes” logic simulation

Conclusion

Formal analysis offers promise to enhance design reset methodologies by providing a deterministic approach to design verification and partial reset implementation using a mix of hardware reset and software initialization. X analysis is able to find X propagation which may be missed by logic simulation at both the RT and gate-levels and provides a quick fix to the problem. X analysis applied at the RTL is much more productive than waiting to find X issues later in the design cycle, where design iterations are substantially more costly to implement.

Reset controllability is another example of X-verification that resolves the problem of not being able to correctly simulate netlists after physical optimizations are applied to restructure the reset logic. By formally analyzing post physical design netlists, simulations can be “fixed” to accurately reflect the reset state of the design.

Related Work

A related application of X-verification is for low power designs with multiple power domains. Here power transition sequences are formally verified to ensure no Xs propagate from powered-down blocks to powered-on blocks and verify correct use of retention and isolation registers. Look for an upcoming white paper on low power formal verification.

Kai-hui Chang is an architect working on Avery's formal product called Insight. He got his Ph. D. from University of Michigan at Ann Arbor and is the recipient of ACM's Outstanding Dissertation Award in EDA at DAC'09. He has 10 years of experience in EDA and has published more than 20 papers in this field.

Chris Browy is co-founder and VP of Sales and Marketing of Avery Design Systems. He is a 25 year veteran of front-end chip design and verification tools and methodologies.