Introduction to Safety Verification
Imagine the following headlines:

- Faulty Mars-Rover Leads To Billion $ Loss
- Death Could Have Been Prevented By Airbag – Massive Callback By Supplier
- Increase of Road-Accidents Due To Faulty Electronic Control Units

You don’t want to make these news!
Why do things break?

Systematic Errors
- Machine Errors
  - Synthesis bugs, ..
- Human Errors
  - Implementation bugs (protocol errors…)
  - Design bugs (broken features, ..)

Random Errors
- Hard Errors
  - Latch-ups, burnouts (stuck-at faults)
- Soft Errors
  - Transients (glitches, bit-flips)

Design Process
- Systematic Errors
  - Minimize!

Physical Effects
- Random Errors
  - Safeguard!

All Devices

Individual Devices
Why Now?

Functional Safety & Standards

- Critical Functionality
- Random Error Probability
- Connected Devices
- Lower Voltage
- Driver Assistance
- Smaller Geometries
- IP Block Increase
- IoT
Introduction to Safety Verification

- Random Errors & Additional Safety Functions
- Safety Standards and Qualification

Requirements Based Verification

- Introduction to Requirements Based Verification
- Faster Quantification of Simulation Test-Benches
- Reliably Exposing Implementation Errors
- Reliable Quantification of Formal Assertion Sets

Verification of Safety Functions

- Efficiently Verifying Redundant Logic

Additional Concerns

- Swiftly Avoiding Coding Errors
- Easier Avoiding Synthesis Bugs
- Saving big on Safety Fault Qualification
Individual devices break in the field because of **Random Errors**

- Random error events **cannot be prevented** → Eventually some normal design function gets corrupted
- However, potential error causes failure only if
  - Error event leads to fault
  - Fault is activated
  - Fault propagates to observable point
  - Fault is observed as erroneous data
- How to prevent problem?

**Safeguard Against Random Errors by Design!**
Random Errors

• Random Errors:
  – Single bit logic errors, such as
    • Stuck-at-0 / Stuck-at-1 fault
    • Glitch or bit-flip
  – Multiple bit logic errors
    • Combinations of the above
    • At multiple locations

• Examples:
  – Radiation Induced Random Errors
  – Hard Errors, such as
    • SEL – Single EventLatch-Up
    • SEB – Single Event Burnout
  – Soft Errors, such as
    • SEU - Single Event Upset
    • SET – Single Event Transient

SINGLE EVENT EFFECTS SPECIFICATION

Single Event Upset (SEU) - a change of state or transient induced by an energetic particle such as a cosmic ray or proton in a device. This may occur in digital, analog, and optical components or may have effects in surrounding interface circuitry (a subset known as Single Event Transients (SETs)). These are "soft" errors in that a reset or rewriting of the device causes normal device behavior thereafter.

Single Hard Error (SHE) - an SEU which causes a permanent change to the operation of a device. An example is a stuck bit in a memory device.

Single Event Latchup (SEL) - a condition which causes loss of device functionality due to a single event induced high current state. An SEL may or may not cause permanent device damage, but requires power strobing of the device to resume normal device operations.

Single Event Burnout (SEB) - a condition which can cause device destruction due to a high current state in a power transistor.

Single Event Gate Rupture (SEGR) - a single ion induced condition in power MOSFETs which may result in the formation of a conducting path in the gate oxide.

Single Event Effect (SEE) - any measurable effect to a circuit due to an ion strike. This includes (but is not limited to) SEUs, SHEs, SELs, SEBs, SEGRs, and Single Event Dielectric Rupture (SEDR).

Multiple Bit Upset (MBU) - an event induced by a single energetic particle such as a cosmic ray or proton that causes multiple upsets or transients during its path through a device or system.

Linear Energy Transfer (LET) - a measure of the energy deposited per unit length as a energetic particle travels through a material. The common LET unit is MeV·cm²/mg of material (Si for MOS devices, etc...).

Threshold LET (LETth) - the minimum LET to cause an effect at a particle fluence of 1E7 ions/cm². Typically, a particle fluence of 1E5 ions/cm² is used for SEB and SEGR testing.

Safeguarding Against Random Errors

- **Fault Detection** by Checkers
  - Raise alarm and enter safe mode (outside correction)
- **Fault Handling** by Redundancy
  - Ensure safe operation on fault detection
  - Correct erroneous data (self correction)
- **Examples**
  - TMR, ECC, lock-step, FSM safe encoding
  - Alarms and error management units
General Flow for High Reliability Applications

Establish project plans and standards
  Define strategy for reliability

Allocate system functions to hardware
  Create derived requirements

High level description of design
  Identify major components

RTL Design, Compile

Synthesis and P & R
  Timing Model
  Generate bitstream file

Program Device / Generate Mask

Planning

Requirements

Conceptual Design

Detailed Design

Implementation

Systematic errors introduced at each step
Prevent systematic errors by process
Random errors are introduced in the field
Safeguard against random errors by design
Reliability functions are tracked through complete flow
Quantification & Qualification of Safety Measures

**Functional Safety Standards**

- Imply rigorous requirements on design and verification
- Governed by strict rules, industry / domain specific
- Failure to comply can result in harm to people, loss of business and prosecution

**Quantification & Qualification** demonstrate effectiveness of safety measures and is required to meet safety standards.
Additional Effort for Functional Safety

Functional Safety & Certification

Minimize Systematic Errors
- Most Rigorous Verification
- Quantification of Verification

Safeguard Random Errors
- Additional Safety Functions
- Qualification of Safety Faults

Additional Effort

Time-to-Market & Budget!
How to get there?

• Minimizing Systematic Errors
• Safeguarding for Random Errors
• Qualification for Certification

Efficiently Supported by

Formal Verification!
Verification Concerns

**RTL Code Quality**
- Enforce best practices on coding
  - Linting/DRC
  - **Formal AutoChecks**

**Requirements Based Verification**
- Ensure requirements are met
  - Simulation/emulation/prototyping
  - **Formal assertion based verification**
- Ensure sufficient verification quality
  - **Quantitative analysis of verification environment**

**Implementation Verification**
- Ensure design tools introduce no systematic errors
  - Tool Certification
  - Independent output assessment, e.g. **equivalence checks**
- Ensure safeguarding against random errors
  - **Qualification of safety faults**
Introduction to Requirements Based Verification
Generic Verification Flow with Requirements Tracing

Requirements

Design

Verification

Quantitative Analysis

Report

Feedback

Requirements
## Functional & Safety Requirements

### Application Scenario: FIFO

**Functional Requirements**

1. The fifo is not full and empty at the same time
2. The fifo is empty after DEPTH many reads without writes
3. The fifo is full after DEPTH many writes without reads
4. The fifo is no longer empty after a write
5. The first data written to an empty fifo leaves the fifo unmodified on the first read

**Safety Requirements**

1. If no error occurs, nothing is flagged and the data is uncorrupted
2. If one error occurs, no error is flagged, the data is uncorrupted and the correction is flagged
3. If two errors occur, an error is flagged, but no correction

---

*Copyright OneSpin Solutions 2015*
Configurable synchronous FIFO buffer, implemented by read and write pointers

```verilog
module fifo #(  
    WIDTH = 8, DEPTH = 32) 
  (clk, reset_n,  
    wr_en, wr_data, full, 
    rd_en, rd_data, empty);

< ... >
logic [DEPTH-1:0][WIDTH-1:0] mem;  
  // FIFO memory
```
Detailed Design
Safeguarding a FIFO with ECC

- **Safety Functions**
  - Detect 1-bit errors and correct them
  - Detect 2-bit errors and raise alarm

- **Design:**
  - Encoder adds e data bits stored in FIFO
  - Decoder detects & corrects 1-bit faults on read \( (\text{error}=0, \text{corrected}=1) \)
  - Decoder detects 2-bit faults on read \( (\text{error}=1) \)
General Coverage Driven Test Framework

Coverage closes test requirements loop

- plan
- coverage metrics
- stimulus generation
- checkers/assertions
- DUV
- report
Simple Simulation Based Verification Flow

Test Plan
Requirements Based

- Requirement 1: R1
  - Test 1: R1.T1
  - Test 2: R1.T2
  - **Test R1.T2**
- Requirement 2: R2
  - Test 2: R2.T1
  - Test 2: R2.T2
  - ..
- ...

Testing
Simulation with Coverage

```
`timescale 1ns/1ns
module TB_R1_T1_test ();
logic sim_okay = 1'b1;
wire clk;
wire rd_en;
```

Test Result
Quantitative Result

Simulation
PASS / FAIL

Cover Points
Line Coverage
Block Coverage

90%
### Application Scenario: FIFO

**Functional Requirements**

- **The fifo is not full and empty at the same time:**
  - check.not_full_and_empty_a

- **The fifo is empty after DEPTH many reads without writes:**
  - check.empty_afterDEPTH_reads_a

- **The fifo is full after DEPTH many writes without reads:**
  - check.full_afterDEPTH_writes_a

- **The fifo is no longer empty after a write:**
  - check.not_empty_after_write_a

- **The first data written to an empty fifo leaves the fifo unmodified on the first read**
  - check.first_data_not_corrupted_a
Step 1:
Cover each requirement with more tests, according to verification plan

Example: check.first_data_not_corrupted_a
"The first data written to an empty fifo leaves the FIFO unmodified on the first read"

Test Result: PASS
Step 2: Validate Test Bench

→ Quantify coverage on design.

Code Coverage Result: 86%
Step 3: Requirements based validation
→ Create cover points for each requirement!

```vhdl
property data_not_corrupted_p;
reg[WIDTH-1:0] dat;
    (empty & wr_en, dat=wr_data[WIDTH-1:0]) ##1
!rd_en[*0:$] ##1 rd_en |=> rd_data[WIDTH-1:0]==dat;
endproperty;
```

Example: `cover.first_data_not_corrupted_c`
“The first data written to an empty fifo leaves the FIFO unmodified on the first read”

Test Coverage Result: COVER_PASS
1. Write tests and cover points for requirements
2. Run tests through simulator
3. Fix any issues you find
4. Feed back coverage to requirements tool

**Example:**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Test Case</th>
<th>Coverage Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>The fifo is not full and empty at the same time:</td>
<td>check.not_full_and_empty_a</td>
<td>PASS, COVER_PASS</td>
</tr>
<tr>
<td>The fifo is empty after DEPTH many reads without writes:</td>
<td>check.empty_afterDEPTH_reads_a</td>
<td>PASS, COVER_PASS</td>
</tr>
<tr>
<td>The fifo is full after DEPTH many writes without reads:</td>
<td>check.full_afterDEPTH_writes_a</td>
<td>PASS, COVER_PASS</td>
</tr>
<tr>
<td>The fifo is no longer empty after a write:</td>
<td>check.not_empty_after_write_a</td>
<td>PASS, COVER_PASS</td>
</tr>
<tr>
<td>The first data written to an empty fifo leaves the fifo unmodified on the first read</td>
<td>check.first_data_not_corrupted_a</td>
<td>PASS, COVER_PASS</td>
</tr>
</tbody>
</table>

**Coverage Result for Simulation:**

86%
What can happen?

**Tests fail?**
- Debug and fix design
- Debug and fix tests

**Code not reached / coverage too low?**
- Add more tests
- **Use Formal Unreachability App**
  - Create a coverage exclude
  - Identify and delete redundant code

**Cover points not hit?**
- Implement more features
- Improve tests

**Too hard to write tests?**
- Add more people
- **Use Formal Assertion Based Verification**
  - Cover more functionality in less time
Faster Qualification of Simulation Test-Benches

Formal Exposes Unreachability
Potential Coverage Problem: Unreachable Code

```plaintext
case (state)
    2'b00: nstate = 2'b01;
    2'b01: nstate = 2'b11;
    2'b10: nstate = 2'b00;
    2'b11: if (ack)
        nstate = 2'b10;
    else
        nstate = 2'b11;
endcase
```

• Use a formal unreachability app
• Unreachable:
  - Proves that code branch cannot be covered
  - May create waivers for coverage tool
  - Note: Careful! Unreachable code often points to DUT issue
• Reachable:
  - Shows simulation trace from reset where code branch gets activated
  - May help improve testbench for hard to reach corner cases
• Both can increase coverage!
Unreachability
How is formal different?

Simulation

- **unreached**
  No simulation trace is available. Could be reachable or unreachable.

- **reached**
  a) by simulation, or
  b) witness for an assertion or cover property, or
  c) witness for statement reachability (generated cover property)

Formal

- **dead (unreachable)**
  Not controllable, no simulation trace exists, *proved dead*.

- **constrained (unreachable)**
  not controllable, no simulation trace exists under the given constraints, *proved dead under constraints*.

- **constraint**
  May exclude certain traces
Why Formal?

Only formal can prove the absence of something!
1. Run formal unreachability app
2. Inspect unreachable code
3. Delete redundant code or create waivers
4. Create testbench stubs for reachable but uncovered code

**Example:**

- **The fifo is not full and empty at the same time:**
  - check.not_full_and_empty_a
  - **PASS, COVER_PASS**

- **The fifo is empty after DEPTH many reads without writes:**
  - check.empty_after_DEPTH_reads_a
  - **PASS, COVER_PASS**

- **The fifo is full after DEPTH many writes without reads:**
  - check.full_after_DEPTH_writes_a
  - **PASS, COVER_PASS**

- **The fifo is no longer empty after a write:**
  - check.not_empty_after_write_a
  - **PASS, COVER_PASS**

- **The first data written to an empty fifo leaves the fifo unmodified on the first read:**
  - check.first_data_not_corrupted_a
  - **PASS, COVER_PASS**

**Coverage Result:** 90%
Formal Unreachability App

OneSpin 360 DV

Cover Property / Activation Check

Formal Check

Witness

Simulation
Testbench Stub

Proven Unreachable

Simulations
Coverage
Exclude

Uncovered Regions

Simulation
Coverage
DB

RTL
Code

HDL Simulator

Test Bench
Example: Quantitative Analysis of Simulation Completeness in DO-254

Level A/B designs governed by DO-254
Appendix B devise coverage on sub-functional level during elemental analysis

... case (state)
  2'b00: nstate = 2'b01;
  2'b01: nstate = 2'b11;
  2'b10: nstate = 2'b00;
  2'b11: if (ack)
    nstate = 2'b10;
  else
    nstate = 2'b11;
endcase
...

Problem

• Insufficient code coverage during DO-254 elemental analysis prevents certification
• Example: Some branch is never executed because ‘ack’ cannot be asserted in context

Solution

• Automatic formal unreachability inspection automatically identifies problems
• Proven dead code can be documented or fixed to achieve certificate
Reliably Exposing Implementation Errors

Formal Assertion-Based Verification
• **Assertion:**
  - Statement of design intent that can be checked in dynamic simulation and formal verification. (*)

• **Examples:**
  - The FIFO must never be full and empty simultaneously

• Can be expressed in different languages: SVA, PSL, OVL, Verilog, VHDL, etc.

---

**Example: Any request is granted within 3 cycles**

<table>
<thead>
<tr>
<th>Active</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
</table>

- **Active:** Assertion is being examined currently
- **Pass:** Simulation trace where assertion occurs as desired (covered)
- **Fail:** Simulation trace where assertion is violated

Formal Assertion Based Verification

- **Early**: No stimuli or testbench is needed
- **Efficient**: Typically check-debug-fix in minutes
- **Exhaustive**: If assertion holds -> no simulation needed
## Application Scenario: FIFO

### Functional Requirements and Assertions

1. Write assertions for requirements
2. Run assertions through formal tool
3. Fix any issues you find
4. Feed back assertion coverage to requirements tool

### Example:

<table>
<thead>
<tr>
<th>Requirement Description</th>
<th>Assertion Code</th>
<th>Cover Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>The fifo is not full and empty at the same time:</td>
<td>check.not_full_and_empty_a</td>
<td>FORMAL_PROOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COVER_PASS</td>
</tr>
<tr>
<td>The fifo is empty after DEPTH many reads without writes:</td>
<td>check.empty_after_DEPTH_reads_a</td>
<td>FORMAL_PROOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COVER_PASS</td>
</tr>
<tr>
<td>The fifo is full after DEPTH many writes without reads:</td>
<td>check.full_after_DEPTH_writes_a</td>
<td>FORMAL_PROOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COVER_PASS</td>
</tr>
<tr>
<td>The fifo is no longer empty after a write:</td>
<td>check.not_empty_after_write_a</td>
<td>FORMAL_PROOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COVER_PASS</td>
</tr>
<tr>
<td>The first data written to an empty fifo leaves the fifo unmodified on the first read</td>
<td>check.first_data_not_corrupted_a</td>
<td>FORMAL_PROOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COVER_PASS</td>
</tr>
</tbody>
</table>
Application Scenario: FIFO
Failing Assertion Exposes RTL Bug

Covered in Simulation
Reliable Quantification of Formal Assertion Sets

Coverage Reloaded
Quantification of Verification to Qualify Verification Environment

Functional Verification

- Simulation / Constraint Random / Emulation
- Formal Assertion Based Verification

- How to know if good enough?
- How to demonstrate that?

Two Dimensions of Quantification:

- **Activation** of Functionality → Quantification of **Input Quality**
- **Observation** of Functionality → Quantification of **Checker Quality**

Qualification according to Safety Standards

Note: We use activation / reachability synonymously…
**Observation Coverage Principle**

- Has the statement been **activated**?
- Idea:
  - If a statement has not been activated during verification, it can’t break a check.
  - If a statement has been reached, would some check fail?
- Can measure quality of **stimulus**.

- Has the effect been **observed**?
- Idea:
  - If a statement is modified and activated, some check should fail.
  - But would some check fail, if the statement cannot be reached?
- Can measure quality of **checkers**.

---

**Example: Statement Coverage**

```
case (state) ...
  burst:
    if (cancel_i)
      done_o <= 1 ...
```

**Been there!**
```
  active
```

**Done that!**
```
  modifydone_o <= v ...
```

---

**Copyright OneSpin Solutions 2015**
Quantitative Analysis of Verification

How much of my DUV is verified? When is my verification finished?

Coverage metrics
- stimulus
- checkers/assertions

DUV

How good are my test vectors & constraints?

Activation Coverage:
- Focused on quality of stimuli
- But what about the checkers?

Observation Coverage:
- Focused on quality of checkers
- Exposes unverified DUV parts

Quantification of verification is required to meet for safety standards!
Cone-of-Influence Coverage

Covered by COI of A

Assertion A

DUV
Observation Coverage Principle
Simple Example

module select1 (onehot, a, b, c, z, clk, reset);

input clk;
input reset;
input [2:0] i;
input a;
input b;
input c;
output reg z;

always @(posedge clk or posedge reset)
if (reset)
z <= 1'b0;
else
begin
   case (i)
      3'b001: z <= a;
      3'b010: z <= b;
      3'b100: z <= c;
      default: z <= 1'b1;
   endcase
end

// if there is no reset, then 'a' is stored in 'z' if 'i' is 3'b001
A: assert property
   (@(posedge clk)
    disable iff (reset)
    i == 3'b001 |=> z == $past(a)
   );
endmodule

Take a statement like this:
3'b001: z <= a;

Change it slightly
3'b001: z <= ??;

This assertion should trigger
A problem in this statement will be detected during verification

However
A lot depends on what change is made to the input and how the coverage is processed:
Quantify Technology
Comparing Coverage Results

• All Assertions:
  – COI Coverage: 100%
  – Quantify Coverage: 70%

• Single Assertion: first_data_not_corrupted_a:
  – COI Coverage: 100%
  – Quantify Coverage: 20%  This looks fishy!

```verilog
property data_not_corrupted_p;
reg[WIDTH-1:0] dat;
  (empty & wr_en, dat=wr_data[WIDTH-1:0]) #1
  !rd_en[*0:$] #1 rd_en |=> rd_data[WIDTH-1:0]==dat
  || (!full | empty);
endproperty;
```
Comparing Coverage Results

• All Assertions:
  – COI Coverage: 100%
  – Quantify Coverage: 70%

• Single Assertion: first_data_not_corrupted_a:
  – COI Coverage: 100%
  – Quantify Coverage: 20%  This looks fishy!

```verilog
property data_not_corrupted_p;
reg[WIDTH-1:0] dat;
  (empty & wr_en, dat=wr_data[WIDTH-1:0]) ##1
  !rd_en[*0:$] ##1 rd_en |=> rd_data[WIDTH-1:0]==dat || (!full | empty); // bad
endproperty;
```

BAD
Stronger Assertion Exposes Bug

Don‘t trust the COIs!
Prover Coverage is not objective!

- Not covered what the prove engine did not need
- Corresponds to abstractions inside prove engines
- Each prove engine uses different abstractions
- Better prove engines give lower coverage!
- Fishy!
Don’t trust the COIs! They are fishy.

Cone-of-Influence (COI) Coverage
- Good to spot big gaps quickly
- Too coarse for sign-off

Prover Coverage
- Result depends on selected prove engine
- Not objective

Mutation Coverage
- High run time (one fault at a time)
- Intrusive, cannot cover all locations

Formal Observation Coverage
- Not intrusive, and objective
- Faster execution (parallel fault injection)

Disqualified for Safety!

Qualified

Best for Safety
Quantification of Formal Verification Environment in ISO-26262

Problem

- Quantitative assessment of formal verification environment needed
- Example: Qualify verification environment for safety functions

Solution

- Use observation coverage to identify coverage holes
- Integrate coverage results with simulation coverage

Example:

“Formal Safety Verification With Qualified Property Sets”
Holger Busch at DAC’14 in Accelerating Productivity Through Formal and Static Methods, Session 38.3
Using Quantify Observation Coverage
Assertion Development & Quantification

- Push-Button solution
- Unique patented technology
- Much more accurate than cone analysis
- Used by multiple customers on their most critical IP

<table>
<thead>
<tr>
<th>Design</th>
<th>#Code Lines</th>
<th>#Assertions</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>321</td>
<td>30</td>
<td>100s</td>
</tr>
<tr>
<td>FSM-DDR2-Read</td>
<td>839</td>
<td>6</td>
<td>106s</td>
</tr>
<tr>
<td>vCore-Processor</td>
<td>295</td>
<td>8</td>
<td>204s</td>
</tr>
<tr>
<td>Arithmetic Block</td>
<td>383</td>
<td>2</td>
<td>257s</td>
</tr>
</tbody>
</table>

Interactive use on single modules to improve verification

Real example at Infineon:
Quantify identified verification holes and guided assertion development.
New assertions detected critical bugs.

Quantify now used to provide management metrics on all designs!

Technology discussed by an end-user at a Test and Verification Seminar, UK, Nov 2013:
Verification of Safety Functions
Safeguarding Against Random Errors

- **Fault Detection** by Checkers
  - Raise alarm and enter safe mode (outside correction)
- **Fault Handling** by Redundancy
  - Ensure safe operation on fault detection
  - Correct erroneous data (self correction)
- **Examples**
  - TMR, ECC, lock-step, FSM safe encoding
  - Alarms and error management units
Efficient Verification of Safety Functions

Fault Injection complexity for \(wd\) data bits and \(we\) redundant bits:

- \(2^{wd}\) possible data input combinations
- \((wd+we)\) 1-bit errors
- \(((wd+we)*(wd+we-1))\) 2-bit errors

Safety Verification Problem

- Safety functions are inactive under normal operation!
- Artifically inject faults into verification to activate

Simulation Based Verification is not a good solution:

- Hard to anticipate all relevant conditions
- Inefficient on large detailed requirement sets
- Hard to deal with huge number of faults + combinations!
- No exhaustive testing feasible

Formal ABV with fault injection
How to formally verify?

Formal Assertion Based Verification of Error Correcting Safety Functions

Methodology

- Capture expected behavior in SV assert properties
- Capture constraints as SV assume properties
- Inject different faults, depending on property to verify

Advantages

- No knowledge of ECC algorithm needed for verification
- High number of parameter and inputs handled
- Easy to specify behavior and inject faults
- Can handle huge number of combinations for potential faults
Detailed Design
Safeguarding a FIFO with ECC

• Safety Functions
  – Detect 1-bit errors and correct them
  – Detect 2-bit errors and raise alarm

• Design:
  – Encoder adds e data bits stored in FIFO
  – Decoder detects & corrects 1-bit faults on read (error=0, corrected=1)
  – Decoder detects 2-bit faults on read (error=1)
Formal ABV with Fault Injection Application Scenario: FIFO

- **General Idea:**
  - Capture safety requirements using assertions
  - Inject faults where they can be observed
  - Depending on fault to be modeled, assume that this input is different from functional value, e.g. Stuck-at 0, stuck-at 1, negated, …, connected

- **For FIFO Example:**
  - Inject Bit-Flip faults at SRAM output
Application Scenario: FIFO
Safety Requirements and Assertions

- **safety.no_error**: If no error occurs, nothing is flagged and the data is uncorrupted.

- **safety.corrected_no_error**: If one error occurs, no error is flagged, the data is uncorrupted and the correction is flagged.

- **safety.error**: If two errors occur, an error is flagged, but no correction.
Application Scenario: FIFO
System Verilog Assertions for Safety Features

No error $\rightarrow$ nothing flagged, data uncorrupted:

\[
\text{no\_error: assert property (disable iff (!reset\_n) empty & wr\_en ##1 rd\_en} \\
|=> \text{rd\_data == $\text{past}(wr\_data,2) & !rd\_error & !rd\_corrected)};
\]

One error $\rightarrow$ no error flagged, data uncorrupted, correction flagged:

\[
\text{corrected\_no\_error: assert property (disable iff (!reset\_n) empty & wr\_en ##1 rd\_en} \\
|=> \text{rd\_data == $\text{past}(wr\_data,2) & !rd\_error & rd\_corrected)};
\]

Two errors $\rightarrow$ error flagged, no correction flagged:

\[
\text{error: assert property (disable iff (!reset\_n) empty & wr\_en ##1 rd\_en} \\
|=> \text{rd\_error && !rd\_corrected)};
\]
Application Scenario: FIFO Fault Injection

- General Idea:
  - Inject faults where they can be observed
  - Depending on fault to be modeled, assume that this input is different from functional value, e.g. Stuck-at 0, stuck-at 1, negated, ..., connected

- For FIFO Example:
  - Inject Bit-Flip faults at SRAM output
Fault Injection App for Formal ABV

• Conveniently use a **Fault Injection App**:

  Fault candidate bits

  ![Diagram](image)

  Formal setup for n-bit faults of desired type

• User can automatically enable different number/kind of faults for individual assertions

• Possible to verify generic assertions like “a 2-bit fault gets detected” for any 2-bit fault by using automated fault injection app with n=2
Using the Fault Injection App

• Supported fault types:
  – Flip
  – Stuck-at-0
  – Stuck-at-1
  – Free

• Tying fault model to assertion:

```python
set_fault_num_and_type_for_assertion
safety.no_error 0 flip

set_fault_num_and_type_for_assertion
safety.corrected_no_error 1 flip

set_fault_num_and_type_for_assertion
safety.error 2 flip
```
corrected_no_error: assert property (disable iff (!reset_n) empty & wr_en #1 rd_en |=> rd_data == $past(wr_data,2) & !rd_error & rd_corrected);
Example: Safeguarding SRAM

**Problem**
- Random failure on SRAM introduced in field
- Example: Wrong value in memory leads to wrongly computing airbag release condition

**Solution**
- Implement error correcting code (ECC) on SRAM
- Verify proper error correction using formal assertion based verification with DV-Verify

**Simulation Based Verification**
- Hard to anticipate all relevant conditions
- Inefficient on large detailed requirement sets

**Formal ABV with Fault Injection**
- No knowledge of ECC algorithm needed for verification
- High number of parameter and inputs handled
- Easy to specify behavior and inject faults
- Can handle huge number of combinations for potential faults
Example: Safeguarding Status Registers

**Problem**
- Random failure introduced by single event upset in field
- Example: Status register value flips for one cycle

**Solution**
- Use redundancy, error detection and error handling in safety management unit
- Use Formal ABV to ensure correct handling of faults and error propagation to SMU

**Simulation with Fault Injection**
- Limited number of faults
- Low degree of automation
- Long run times

**Formal ABV with Fault Injection**
- Exhaustive fault activation including multiples
- Efficient modeling of faults and specification of requirements
- Highly automated
- Handling large blocks and huge number of faults simultaneously

*) ISO26262 demands verification of safety features on net list level for ASIL-D compliance

(H. Busch, Infineon Technologies, "Formal Safety Verification of Automotive Microcontroller Parts", ZuE2012, Bremen)
Summary on Safety Features

- Random errors cannot be avoided
- Safety functions are added to handle random errors
  - Normally inactive, inject faults into verification to activate
- Simulation intractable
  - Vast number of possible fault locations/combinations to verify

- Formal Verification
  - Efficient modeling of faults and specification of requirements
  - Exhaustive fault activation including multiples
  - Highly automated & fast solution
  - Additional qualification of verification environment

Real Example

- Verifying Safeguarding Mechanisms of Automotive Micro Controller.
- ECC-Correction of single bit errors for 128 bit data words proven in 5 min.
- High automation achieved by generic properties and flow automation.
- Different kinds of bugs detected, e.g. incomplete alarm reduction.
Summary
Safety Critical Verification
Handling the Demands of High Reliability

Meeting tough standards: Rigorous verification of design and safety features

- Exhaustive functional verification
- Reliable quantitative analysis
- Efficient safety verification
- Independent fault qualification

Minimize Systematic Errors
- Most Rigorous Verification
- Quantification of Verification

Safeguard Random Errors
- Additional Safety Functions
- Qualification of Safety Faults

Required by Safety Standards
Where Formal Can Help

Verification

- Easier verification of synthesis results and avoiding synthesis bugs by formal equivalence checking
- Swiftly avoiding coding errors by automatic formal inspection
- Reliably exposing implementation errors by formal assertion based verification
- Efficiently verifying additional safety functions using formal fault injection

Qualification

- Faster qualification of simulation test-benches by formal reachability analysis
- Reliably quantifying formal assertion sets by formal observation coverage metrics
- Saving big on safety fault qualification by formal safety fault analysis
Why Formal Verification?
Everybody Wins

Formal verification is an automatic solution for proving what is often done manually. Implementing a software based formal verification process implies less availability to systematic failure and random failure.

Increase the quality of your process standards while speeding up implementation.
Want more?

To learn more about safety critical design & verification:

• Read Safety Critical News
  – http://safetycritical.onespin-solutions.com/

• Visit us at Booth #3126