Hybrid Linting:

An efficient method to overcome challenges with Structural Linting in Arithmetic Overflow Verification

Esra Sahin Basaran and Kai-wen Chin
Renesas Electronics

Agenda

- Arithmetic Overflow Verification Challenge
- Arithmetic Logic Categories
- Formal LINT for Each Category
- Pitfalls in Explicit Signed Logic
- Conclusion

Arithmetic Overflow Verification Challenge

Arithmetic Overflow Verification Challenge

```
input  [3:0] A;
input  [3:0] B;
output [3:0] Y;

assign Y[3:0] = A[3:0] + B[3:0];
```

- RHS width is 5-bit, including carry
- LHS width 4-bit
- LHS is not wide enough → Overflow!

- Arithmetic overflow verification:
  - Unsigned arithmetic
  - Signed arithmetic
- Traditional methods can be inefficient:
  - **Dynamic simulation:** Hard to be exhaustive
  - **Structural LINT:** Lots of false negatives

Arithmetic Overflow Verification Challenge

- Formal LINT = Structural LINT + Formal Verification
  - Auto-generated SystemVerilog Assertions for Formal Verification
- Formal LINT looks promising
  - But…
  - The presentation provides its prerequisite

Arithmetic Logic Categories

Arithmetic Logic Category

- Unsigned Logic
  - The design implements unsigned arithmetic
  - No “signed” keyword
  - “Part select syntax” could be used for readability
  - Manual “zero-padding” at MSBs could be used for readability
- Example

```
wire [3:0] FUL_U1A, FUL_U1B; // Variables are intended to be unsigned.
wire [4:0] Y_U1A = FUL_U1A + FUL_U1B;
wire [4:0] Y_U1B = FUL_U1A[3:0] + FUL_U1B[3:0];
wire [4:0] Y_U1C = {1'b0,FUL_U1A[3:0]} + {1'b0,FUL_U1B[3:0]};
wire [4:0] Y_U1D = {1'b0,FUL_U1A} + {1'b0,FUL_U1B};
```

Arithmetic Logic Category

- Explicit Signed Logic
  - The design implements signed arithmetic
  - Signed variables are declared using “signed” keyword
  - No part-select syntax for explicit signed variables
  - No manual sign-extension for explicit signed variables
- Example

```
wire        [3:0] FUL_U1A; // Variable is intended to be unsigned
wire        [3:0] FUL_U1B; // Variable is intended to be unsigned
wire signed [3:0] FUL_S1C; // Variable is intended to be signed and declared explicitly
wire signed [3:0] FUL_S1D; // Variable is intended to be signed and declared explicitly

wire signed [4:0] Y_S1A = FUL_U1A + FUL_U1B;
wire signed [4:0] Y_S1B = FUL_S1C + FUL_S1D;
```

Arithmetic Logic Category

- Implicit Signed Logic
  - The design implements signed arithmetic
  - No “signed” keyword. Variable signedness is implied by its consuming logic.
  - Manual sign-extension for implicit signed variable must be used for correctness
  - Part-select syntax could be used for readability
  - Manual zero-padding at MSBs could be used for readability (for unsigned variables)
- Example

```
wire [3:0] FUL_U1A; // Variable is intended to be unsigned
wire [3:0] FUL_S1B; // Variable is intended to be signed but not declared explicitly
wire [4:0] Y_S1A = FUL_U1A + {FUL_S1B[3],FUL_S1B};
wire [4:0] Y_S1B = {1'b0,FUL_U1A} + {FUL_S1B[3],FUL_S1B};
wire [4:0] Y_S1C = {1'b0,FUL_U1A[3:0]} + {FUL_S1B[3],FUL_S1B[3:0]};
```

Formal LINT for Each Category

Formal LINT for Unsigned Logic

- HLF_U1A, HLF_U1B and Y_U3A are all unsigned 4-bit → Overflow?!
- HLF_U1A and HLF_U1B both reduced to half range by logic
- Formal LINT proves Y_U3A has no overflow issue

```
output [3:0] Y_U3A;
input  [3:0] FUL_U1A, FUL_U1B;
wire   [3:0] HLF_U1A = (FUL_U1A > 7) ? 7 : FUL_U1A;
wire   [3:0] HLF_U1B = (FUL_U1B > 7) ? 7 : FUL_U1B;
assign Y_U3A = HLF_U1A + HLF_U1B;
```

LHS value range : 15~0
RHS value range : 7~0 + 7~0 = 14~0
Structural LINT : Violation
Formal LINT : Proven

Value ranges are clamped to test the behavior difference between Structural LINT and Formal LINT.

Formal LINT for Explicit Signed Logic

- All variables declared “signed” explicitly
- HLF_S1A, HLF_S1B and Y_S3F are all signed 4-bit → Overflow?!
- HLF_S1A and HLF_S1B both reduced to half range by logic
- Formal LINT proves Y_S3F has no overflow issue

```
output signed [3:0] Y_S3F;
input  signed [3:0] FUL_S1A, FUL_S1B;
wire   signed [3:0] HLF_S1A, HLF_S1B;
assign HLF_S1A = (FUL_S1A > 3) ? 3 : (FUL_S1A < -4) ? -4 : FUL_S1A;
assign HLF_S1B = (FUL_S1B > 3) ? 3 : (FUL_S1B < -4) ? -4 : FUL_S1B;
assign Y_S3F = HLF_S1A + HLF_S1B;
```

LHS value range : 7~-8
RHS value range : 3~-4 + 3~-4 = 6~-8
Structural LINT : Violation
Formal LINT : Proven

Formal LINT for Implicit Signed Logic

- All variables are intended to be signed but aren't explicitly declared as signed
- The design is correct because HLF_S1e and HLF_S1f are reduced to half range
- Formal LINT treat both operands as unsigned and flag error

```
output [3:0] Y_SCf;
input  [3:0] FUL_S1e, FUL_S1f;
wire   [3:0] HLF_S1e = (FUL_S1e[3:2]==2'b01) ? 4'b0011 :
                       (FUL_S1e[3:2]==2'b10) ? 4'b1100 : FUL_S1e[3:0] ;
wire   [3:0] HLF_S1f = (FUL_S1f[3:2]==2'b01) ? 4'b0011 :
                       (FUL_S1f[3:2]==2'b10) ? 4'b1100 : FUL_S1f[3:0] ;
assign Y_SCf[3:0] = {HLF_S1e[3], HLF_S1e[3:0]} + {HLF_S1f[3], HLF_S1f[3:0]};
```

Intended LHS value range : 7~-8
Intended RHS value range : 3~-4 + 3~-4 = 6~-8

Formal LINT doesn't know the operands are signed in the design intention.
LHS value range : 15~0
RHS value range : 31~0 + 31~0 = 62~0
Structural LINT : Violation
Formal LINT : Violation

Formal LINT is not for all of them!

The key issue is variable's signedness information

Category	Pitfalls	Formal LINT limitation
Implicit Signed Logic	None	• Limitation: Lack of variable signedness information • Formal LINT currently may not accurately analyze • Work-in-progress
Explicit Signed Logic	Many	• No showstopper for Formal LINT • Complementary checks required → **Hybrid LINTing**
Unsigned Logic	None	• No limitation • Formal LINT is fully capable

Pitfalls in Explicit Signed Logic

Pitfall in Explicit Signed Logic

- Verilog-2001 and 2005 defined syntax for signed arithmetic.
- However, designers must be aware of some rules to avoid incorrect design.
- Refer to Verilog-2005 LRM:
  - Section 3.5.1 Integer constants
  - Section 5.1.2 Binary operator precedence
  - Section 5.1.3 Using integer numbers in expressions
  - Section 5.1.6 Arithmetic expressions with regs and integers
  - Section 5.1.7 Relational operators
  - Section 5.1.8 Equality operators
  - Section 5.1.12 Shift operators
  - Section 5.4 Expression bit lengths
  - Section 5.5 Signed expressions

Pitfall 1: Signed-to-unsigned conversion

Due to mixture of signed and unsigned in expression

Example:

```
wire [3:0] FUL_U1A;
wire signed [3:0] FUL_S1A;
wire signed [3:0] FUL_S1B;

wire [4:0] Y_U1C = FUL_S1A + FUL_S1B;

wire [4:0] Y_U1B = FUL_U1A + FUL_S1A;
```

Mixture of signed and unsigned

Unsigned variable

Signed variable converted into unsigned

Pitfall 1: Signed-to-unsigned conversion

Due to mixture of signed and unsigned in expression

Example:

```
wire          [3:0] A_U4b;
wire signed   [3:0] A_S4b;
wire signed   [3:0] B_S4b;
wire signed   [4:0] Y_S5b = A_S4b + B_S4b;

wire signed   [4:0] Z_S5b = A_U4b + B_S4b;
```

```
A_U4b = +4 = 4'b0100
B_S4b = -1 = 4'b1111
Expected Z_S5b = +3
```

B_S4b is converted to unsigned

```
Z_S5b[4:0] = A_U4b + B_S4b
           = 5'b00100 + 5'b01111
           = 5'b10011
           = -13
```

What's the impact?

```
A_S4b = +4 = 4'b0100
B_S4b = -1 = 4'b1111
Expected Y_S5b = +3

Y_S5b[4:0] = A_S4b + B_S4b
           = 5'b00100 + 5'b11111
           = 5'b00011
           = +3
```

Pitfall 1: Signed-to-unsigned conversion

Due to mixture of signed and unsigned in expression

What's the impact?

Example:

```
wire        [3:0] U4b_0 = 4'b0100; // +4
wire signed [3:0] S4b_0 = 4'b0100; // +4
wire signed [3:0] S4b_1 = 4'b1111; // -1

wire Y_0 = (S4b_0 >= S4b_1); // Y_0 = 1 (simulation result)

wire Y_1 = (U4b_0 >= S4b_1); // Y_1 = 0 (simulation result)
```

U4b_0 is interpreted as +4

S4b_1 is interpreted as +15 because it is converted into unsigned

Pitfall 2: Signed-to-unsigned conversion

Due to concatenation

- Signed variable will be automatically sign-extended.
- However, manual sign-extension will cause signed-to-unsigned conversion

Example:

```
wire signed [3:0] S4b_A;
wire signed [3:0] S4b_B;
wire signed [4:0] S5b_Y;
wire signed [4:0] S5b_Z;
assign S5b_Y = S4b_A + S4b_B;
assign S5b_Z = {S4b_A[3],S4b_A} + {S4b_B[3],S4b_B};
```

Result of **concatenation** is treated as unsigned

Pitfall 3: Signed-to-unsigned conversion

Due to part-select

Example:

```
wire signed [3:0] S4b_A;
wire signed [3:0] S4b_B;
wire signed [4:0] S5b_X;
wire signed [4:0] S5b_Y;
wire signed [4:0] S5b_Z;
assign S5b_X = S4b_A + S4b_B;
assign S5b_Y = S4b_A[3:1] + S4b_B[3:1];
assign S5b_Z = S4b_A[3:0] + S4b_B[3:0];
```

Part-select changes variable into unsigned

They will be automatically zero-padded instead of sign-extended

Pitfall 4: Sign Casting

- To avoid signed-to-unsigned conversion, we can use $signed().
- However, pitfall again…

Example:

```
wire [3:0] A_U4b;
wire signed [3:0] B_S4b;
wire signed [5:0] X_S6b = A_U4b + B_S4b; // Signed-to-unsigned conversion

wire signed [5:0] Y_S6b = $signed(A_U4b) + B_S4b; // Bad sign casting

wire signed [5:0] Z_S6b = $signed({1'b0,A_U4b}) + B_S4b; // Good sign casting
```

Example: If A_U4b is "+15", $signed(A_U4b) will be interpreted as "-1"

Solution: Zero-padding before sign-casting

Pitfall 5: Signed Constant

Example:

```
wire signed [3:0] A_S4b;

wire signed [5:0] X_S6b = A_S4b + 4'd8;

wire signed [5:0] Y_S6b = A_S4b + 4'sd8;

wire signed [5:0] Z_S6b = A_S4b + 5'sd8;
```

4'd8 is unsigned.
A_S4b is converted into unsigned due to mixture of signed and unsigned variables.

Use a signed constant to keep A_S4b as signed.
"4'sd8" is actually "-8" as a signed number.

This is the right way to do it!

Value range of 4-bit signed constant : +7 ~ -8
Value range of 5-bit signed constant : +15 ~ -16

Pitfall 6: Interim Result overflow

Example:

```
wire [3:0] B = 4'b0011;
wire [3:0] C = 4'b1110;

                          // Simulation result:
wire [3:0] Y2 = (B+C) >>> 1; // Y2 = 4'b0000

wire [3:0] Y4 = (B+C) / 2  ; // Y4 = 4'b1000
```

"B+C" evaluated as 4-bit expression → Overflow already before shift

Constant "2" is 32-bit. "(B+C)/2" is evaluated as 32-bit expression. Result is correct.

- Refer to Verilog-2005 LRM:

Table 5-22—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined

Solution: VC SpyGlass

Check Items		LINT Type	VC SpyGlass Coverage	SpyGlass Rule
Signed-to-unsigned conversion	Due to mixed signed and unsigned operands	Structural	Covered	SignedUnsignedExpr-ML
	Due to part-select	Structural	Covered	SignedUnsignedConvert-ML
	Due to concatenation	Structural	Covered	SignedUnsignedConvert-ML
Bad sign-casting		**Formal**	Covered	SignedSysFuncUsage-ML
Bad signed constant		Structural	Covered	LiteralUnderflow-ML LiteralOverflow-ML
Interim result overflow		**Formal**	**Part of Roadmap**	
Arithmetic overflow		**Formal**	Covered	SignedUnsignedExpr-ML W164a NegativeValueInfer-ML W110

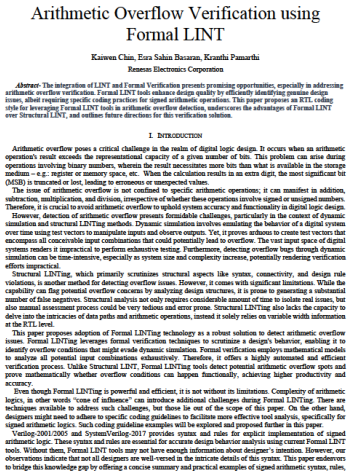
Hybrid Linting

Conclusion

Conclusion

- Formal LINT is not efficient for **Implicit Signed Logic**
- Formal LINT is a promising verification solution for **Unsigned Logic** and **Explicit Signed Logic**
- Our DvCon Paper covered details of the pitfalls as well

DVCON2024
Poster (#1020)


Arithmetic Overflow Verification using Formal LINT

Kaiwen Chin, Esra Sahin Basaran, Kranthi Pamarthi

Renesas Electronics Corporation

Abstract— The integration of LINT and Formal Verification presents promising opportunities, especially in addressing arithmetic overflow verification. Formal LINT tools enhance design quality by efficiently identifying genuine design issues, albeit requiring specific coding practices for signed arithmetic operations. This paper proposes an RTL coding style for leveraging Formal LINT tools in arithmetic overflow detection, underscores the advantages of Formal LINT over Structural LINT, and outlines future directions for this verification solution.

I. INTRODUCTION

Arithmetic overflow poses a critical challenge in the realm of digital logic design. It occurs when an arithmetic operation's result exceeds the representational capacity of a given number of bits. This problem can arise during operations involving binary numbers, wherein the result necessitates more bits than what is available in the storage medium – e.g. register or memory space, etc. When the calculation results in an extra digit, the most significant bit (MSB) is truncated or lost, leading to erroneous or unexpected values.

The issue of arithmetic overflow is not confined to specific arithmetic operations; it can manifest in addition, subtraction, multiplication, and division, irrespective of whether these operations involve signed or unsigned numbers. Therefore, it is crucial to avoid arithmetic overflow to uphold system accuracy and functionality in digital logic design.

However, detection of arithmetic overflow presents formidable challenges, particularly in the context of dynamic simulation and structural LINTing methods. Dynamic simulation involves emulating the behavior of a digital system over time using test vectors to manipulate inputs and observe outputs. Yet, it proves arduous to create test vectors that encompass all conceivable input combinations that could potentially lead to overflow. The vast input space of digital systems renders it impractical to perform exhaustive testing. Furthermore, detecting overflow bugs through dynamic simulation can be time-intensive, especially as system size and complexity increase, potentially rendering verification efforts impractical.

Structural LINTing, which primarily scrutinizes structural aspects like syntax, connectivity, and design rule violations, is another method for detecting overflow issues. However, it comes with significant limitations. While the capability can flag potential overflow concerns by analyzing design structures, it is prone to generating a substantial number of false negatives. Structural analysis not only requires considerable amount of time to isolate real issues, but also manual assessment process could be very tedious and error prone. Structural LINTing also lacks the capacity to delve into the intricacies of data paths and arithmetic operations, instead it solely relies on variable width information at the RTL level.

This paper proposes adoption of Formal LINTing technology as a robust solution to detect arithmetic overflow issues. Formal LINTing leverages formal verification techniques to scrutinize a design's behavior, enabling it to identify overflow conditions that might evade dynamic simulation. Formal verification employs mathematical models to analyze all potential input combinations exhaustively. Therefore, it offers a highly automated and efficient verification process. Unlike Structural LINT, Formal LINTing tools detect potential arithmetic overflow spots and prove mathematically whether overflow conditions can happen functionally, achieving higher productivity and accuracy.

Even though Formal LINTing is powerful and efficient, it is not without its limitations. Complexity of arithmetic logics, in other words "cone of influence" can introduce additional challenges during Formal LINTing. There are techniques available to address such challenges, but those lie out of the scope of this paper. On the other hand, designers might need to adhere to specific coding guidelines to facilitate more effective tool analysis, specifically for signed arithmetic logics. Such coding guideline examples will be explored and proposed further in this paper.

Verilog-2001/2005 and SystemVerilog-2017 provides syntax and rules for explicit implementation of signed arithmetic logic. These syntax and rules are essential for accurate design behavior analysis using current Formal LINT tools. Without them, Formal LINT tools may not have enough information about designer's intention. However, our observations indicate that not all designers are well-versed in the intricate details of this syntax. This paper endeavors to bridge this knowledge gap by offering a concise summary and practical examples of signed arithmetic syntax, rules,

DVCON2024
Paper (#1020)

References

[1] "IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[2] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595

[3] Dr. Greg Tumbush, “Signed Arithmetic in Verilog 2001 – Opportunities and Hazards,” in DVCON 2005

Questions?


THANK YOU
Our Technology, Your Innovation™