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,

and potential pitfalls based on Verilog-2001/2005. Familiarity with these intricacies is paramount when employing Formal LINT tools for arithmetic overflow verification.

In this paper, we categorize different RTL coding styles based on their compatibility with Formal LINT tools. We will illustrate these categories with examples that showcase the limitations of Structural LINT tools and the advantages of their Formal LINT counterparts. Additionally, we will present a set of arithmetic RTL coding guidelines to ensure both correct code functionality and effective analysis by Formal LINT tools. Lastly, we will provide a summary of our evaluation of Formal LINT tools in the context of arithmetic overflow verification, offering insights into their effectiveness and potential limitations.

II. GLOSSARY

Glossary	Description
Dynamic simulation	It is a verification method that relies heavily on simulation patterns. Verification coverage is directly related to the completeness of simulation patterns.
Explicit sign extension	It is a coding style where the variable's MSB is explicitly repeated to form a wider variable with same signedness. For example: {A[3], A[3:0]} {A[3], A[3], A[3:0]} {{2{A[3]}}, A[3:0]}
Explicit signed variable	Variable is explicitly declared as signed.
Formal LINT	It is a methodology that combines Structural LINT with Formal Verification technology. In Formal LINT, it detects potential design risks structurally, but then automatically generates Formal Verification rules (assertions) to prove functionally whether the risks are real issues or not if structurally it is possible to make that judgement
Implicit sign extension	Automatic sign extension is executed by simulator and synthesizer based on the context and the rules in Verilog and SystemVerilog LRM. Implicit sign extension means a coding style in which the variable can be automatically extended based on the variable's signedness. If the variable is unsigned, it would be automatically extended by adding 0's to its MSBs. If the variable is declared as signed and there is no signed-to-unsigned conversion, the variable would be automatically extended by repeating its MSB.
Implicit signed variable	Variable is declared with default signedness "unsigned". However, it is used as a signed variable in the context. For example, it is sign-extended like this: {A[3], A[3:0]}
LHS	Left hand side
RHS	Right hand side
Structural LINT (Or simply "LINT")	It is a methodology that performs analysis on RTL code structure to conservatively report potential risks that could lead to design failures and issues in implementation flow. It analyzes RTL code structurally but not functionally, so it tends to generate false negatives.

III. CODING STYLES FOR ARITHMETIC LOGICS

In this paper, we focus on three example coding styles for arithmetic logics.

A. Unsigned Arithmetic Logics

Example1:

```
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};
```

All variables in "Example1" are unsigned. "Y_U1A~D" implements the same equation with slightly different coding styles. This coding style includes following features:

- The design implements unsigned arithmetic
- No "signed" keyword
- Explicit part select syntax could be used for readability
- Explicit MSB zero-padding could be used for readability

B. Implicit Signed Variable Coding Style

Example2:

```
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]};
```

All variables are declared as unsigned in "Example2". However, "FUL_S1B" is a signed variable in designer's intention. Therefore, FUL_S1B is manually sign-extended to implement correct logics. This coding style includes following features:

- The design implements signed arithmetic
- No "signed" keyword
- Explicit part-select syntax could be used for readability
- Explicit MSB zero-padding could be used for readability (for unsigned variables)
- Explicit sign-extension for implicit signed variable must be used for correctness

C. Explicit Signed Variable Coding Style

Example3:

```
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;
```

In "Example3", signed variables are explicitly declared and the variable declarations match designer's intentions. "Y_S1A" is an explicit signed variable and assigned to an unsigned result from RHS equation. "Y_S1B" is an explicit signed variable and is assigned to a signed result from RHS equation. This coding style includes following features:

- The design implements signed arithmetic
- Signed variables are declared using "signed" keyword
- No part-select syntax for explicit signed variables
- No explicit sign-extension for explicit signed variables

IV. LIMITATIONS OF STRUCTURAL LINT

LINT tools have been supporting RTL designers to structurally detect design issues early in the design cycle. One of the key problems they seek to address is "arithmetic overflow". Arithmetic overflow reported by structural LINT tools might inevitably have "false negatives", giving designers a hard time to isolate real issues. Manual assessment of certain types of violations can be tedious, error-prone, and very time-consuming.

Some successful and false negative examples are given below.

A. Successful examples

No violation reported with structural verification in "Example4". "FUL_U1A" and "FUL_U1B" are both unsigned 4-bit. Their sum yields unsigned 5-bit result and is assigned to an unsigned 5-bit LHS variable. The implementation is structurally correct. Structural verification is sufficient and accurate and formal-aware verification is not required.

"Example5" is like "Example4", except it implements signed arithmetic and is structurally correct. Therefore, formal-aware verification is not required, and Structural LINT is fully capable of analyzing this case.

Example4:

```
input [3:0] FUL_U1A;
input [3:0] FUL_U1B;
output [4:0] Y_U1A;
assign Y_U1A = FUL_U1A + FUL_U1B; // u5 (31 ~ 0) = u4+u4 (30 ~ 0)
```

Example5:

```
input signed [3:0] FUL_S1A;
input signed [3:0] FUL_S1B;
output signed [4:0] Y_S1F,
assign Y_S1F = FUL_S1A + FUL_S1B; // s5 (15 ~ -16) = s4+s4 (14 ~ -16)
```

B. False negative examples

In "Example6", "HLF_U1A" and "HLF_U1B" are both unsigned 4-bit. Their sum is unsigned 5-bit but assigned to an unsigned 4-bit LHS variable. Structurally, it appears to be an incorrect implementation. Structural LINT will report this case as a violation. However, the driving logics of "HLF_U1A" and "HLF_U1B" reduce their value range by

half. So functionally, their sum is 4-bit and the 4-bit LHS variable is wide enough to hold all the possible result from RHS expression. “Example7” is similar to “Example6” except it implements signed arithmetic. Both examples illustrate limitations of Structural LINT’s. Structural LINT only checks the design “structurally” instead of performing “functional” verification.

Example6:

```
input [3:0] FUL_U1A;
input [3:0] FUL_U1B;
output [3:0] Y_U3A;
wire [3:0] HLF_U1A;
wire [3:0] HLF_U1B;
assign HLF_U1A = (FUL_U1A > 7) ? 7 : FUL_U1A; // value range : 7 ~ 0
assign HLF_U1B = (FUL_U1B > 7) ? 7 : FUL_U1B; // value range : 7 ~ 0
assign Y_U3A = HLF_U1A + HLF_U1B; // u4 (15 ~ 0) = u3+u3 (14 ~ 0)
```

Example7:

```
input signed [3:0] FUL_S1A;
input signed [3:0] FUL_S1B;
output signed [3:0] Y_S3F;
wire signed [3:0] HLF_S1A;
wire signed [3:0] HLF_S1B;
assign HLF_S1A = (FUL_S1A > 3) ? 3 : (FUL_S1A < -4) ? -4 : FUL_S1A; // value range : 3 ~ -4
assign HLF_S1B = (FUL_S1B > 3) ? 3 : (FUL_S1B < -4) ? -4 : FUL_S1B; // value range : 3 ~ -4
assign Y_S3F = HLF_S1A + HLF_S1B; //s4 ( 7 ~ -8) = s3+s3 ( 6 ~ -8)
```

V. Formal LINT Advantage and Limitation

LINT combined with Formal technology offers RTL designers exciting new opportunities. EDA vendors claim that Formal LINT tools provide a better solution for signed arithmetic overflow verification. However, efficient formal-aware verification requires RTL designers to adopt a certain coding style.

One limitation about Formal LINT for signed arithmetic logic is about the signedness of the operands. In Implicit Signed Variable Coding Style, all operands are declared as unsigned. Formal LINT tools may have difficulty recognizing which variables are signed in the designer’s intention. Without accurate signedness information of operands, Formal LINT tools cannot report accurate results. On the other hand, designers declare signed operands explicitly in Explicit Signed Variable Coding Style. Formal LINT tools have signedness information for all the operands and are supposed to deliver accurate results. But of course, Formal LINT would still have the same challenges as Formal Verification, such as the complexity of the “cone of influence”.

To accurately detect overflow for signed arithmetic logics, we recommend Formal LINT tools to conduct the analysis from “value range” point of view. An example algorithm is like this:

- Determine LHS variables value range base on its signedness and width.
- Automatically generate assertion rules to validate the RHS equation's result within the LHS variable's value range.
- During RHS equation’s evaluation, the tool must follow LRM’s definition about operand signedness, signed-to-unsigned conversion rules, operand width extension rules, etc.

We have been partnering with EDA tool vendors to make arithmetic overflow verification work properly in Formal LINT tools as we aim to achieve higher productivity and accuracy. Although there are still some pending issues, this paper recommends the most promising RTL coding style if one wants to use Formal LINT to detect arithmetic overflow. Please see Section 6 for details.

A. Successful examples in Unsigned Arithmetic Coding Style

In the “Example8”, "HLF_U1A” and “HLF_U1B” are structurally 4-bit, but their value ranges are functionally reduced by half. “Y_U3A” is 4-bit and Structural LINT will report arithmetic overflow for it. Formal-aware lint would prove there is no overflow for “Y_U3A”.

Example8:

```
input [3:0] FUL_U1A;
input [3:0] FUL_U1B;
output [3:0] Y_U3A;
wire [3:0] HLF_U1A;
wire [3:0] HLF_U1B;
assign HLF_U1A = (FUL_U1A > 7) ? 7 : FUL_U1A; // value range : 7 ~ 0
assign HLF_U1B = (FUL_U1B > 7) ? 7 : FUL_U1B; // value range : 7 ~ 0
assign Y_U3A = HLF_U1A + HLF_U1B; // u4 (15 ~ 0) = u3+u3 (14 ~ 0)
```

B. Successful examples in Explicit Signed Arithmetic Coding Style

While Structural LINT would report overflow at "Y_S3F" in "Example9", Formal LINT proves there is no overflow functionally.

Example9:

```
input signed [3:0] FUL_S1A;
input signed [3:0] FUL_S1B;
output signed [3:0] Y_S3F;
wire signed [3:0] HLF_S1A;
wire signed [3:0] HLF_S1B;
assign HLF_S1A = (FUL_S1A > 3) ? 3 : (FUL_S1A < -4) ? -4 : FUL_S1A; // value range : 3 ~ -4
assign HLF_S1B = (FUL_S1B > 3) ? 3 : (FUL_S1B < -4) ? -4 : FUL_S1B; // value range : 3 ~ -4
assign Y_S3F = HLF_S1A + HLF_S1B; //s4 (7 ~ -8) = s3+s3 (6 ~ -8)
```

C. Limitation of Formal LINT in Implicit Signed Variable Coding Style

In "Example10", "HLF_S1"e and "HLF_S1f "are both declared with default signedness "unsigned". In designer's intention, they are signed variables and they are functionally reduced to half value range 3 ~ -4. In the addition equation, they are explicitly sign-extended and the result is assigned to a 4-bit variable "Y_SCf". Notice that "Example10" is a functionally correct design using Implicit Signed Variable Coding Style. "Y_SCf" won't suffer from arithmetic overflow because the signed value range of the RHS equation is 6 ~ -8, and the 4-bit variable "Y_SCf" can represent a value range of 7 ~ -8.

However, if we use Formal LINT to analyze the same example, "HLF_S1e" and "HLF_S1f" would be treated as unsigned variables since they are not declared as signed. From Formal LINT's point of view, their value ranges would be 15 ~ 0, instead of 3 ~ -4. The addition equation's range would be analyzed to be 62 ~ 0. The equation's result is then assigned to a 4-bit variable "Y_SCf", which can only represent an unsigned value range 15 ~ 0. "Example10" will be reported as an arithmetic overflow in Formal LINT while it is a correct design from designer's point of view.

"Example10" illustrates limitations when we try to use Formal LINT to analyze Implicit Signed Variable Coding Style. Formal LINT could inevitably generate false negatives because it doesn't have signedness information in the design intention.

Example10:

```
input  [3:0] FUL_S1e, // intended to be signed, but not declared so
input  [3:0] FUL_S1f, // intended to be signed, but not declared so
output [3:0] Y_SCf,   // intended to be signed, but not declared so

// HLF_S1e[3:0] value range : 3 ~ -4
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] ;
// HLF_S1f[3:0] value range : 3 ~ -4
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] ;
// s4 (7 ~ -8) = s3+s3 (6 ~  -8)
assign Y_SCf[3:0] = {HLF_S1e[3], HLF_S1e[3:0]} + {HLF_S1f[3], HLF_S1f[3:0]};
```

VI. GUIDELINES FOR EXPLICIT SIGNED VARIABLE CODING STYLE

For unsigned arithmetic logics, Formal LINT tools have all information about designer's intention so they should be able to analyze the design properly. For signed arithmetic logics, we observed two coding styles:

- Implicit Signed Variable Coding Style, and
- Explicit Signed Variable Coding Style

Formal LINT tools are still under development to handle Implicit Signed Variable Coding Style due to previously mentioned limitation. Therefore, we currently recommend Explicit Signed Variable Coding Style so that Formal LINT tools can verify the design correctly.

Unfortunately, there are some pitfalls and know-hows in Explicit Signed Variable Coding Style, so designers must be aware of some guidelines and techniques. We will provide detailed information in the following sections.

A. Signed arithmetic related rules in Verilog-2005

Although Verilog-2001 is famous and commonly referred to, we have concluded Verilog-2005 has more comprehensive information on signed arithmetic RTL coding. Therefore, we have chosen Verilog-2005 LRM to be our main reference in this section. The information and guidelines provided in following sections of the paper are

based on these sections of Verilog-2005 LRM: Section 3.5.1, Section 5.1.2, Section 5.1.3, Section 5.1.6, Section 5.1.7, Section 5.1.8, Section 5.1.12, Section 5.4, Section 5.5.

Simply speaking, designers must pay attention to two main things in signed arithmetic RTL:

- Variable signedness
- Expression width

Following guidelines and techniques are all related to those two fundamental aspects.

B. Signed-to-unsigned Conversion

In Explicit Signed Variable Coding Style, designers explicitly declare signed variables. However, there are circumstances where a signed variable is converted into an unsigned variable in the RHS expression of an assignment. This is usually undesirable and potentially leads to design bugs.

Designers must be careful about following design scenario that cause signed-to-unsigned conversion:

- ***Explicit signed variable is converted into unsigned when it is used in an expression that has any unsigned operand (variable or constants).***

  For example:

```
wire         [3:0] FUL_U1A;
wire  signed [3:0] FUL_S1A;
wire  signed [3:0] FUL_S1B;
wire         [4:0] Y_U1B = FUL_U1A + FUL_S1A;  // Example12
wire         [4:0] Y_U1C = FUL_S1A + FUL_S1B;  // Example13
wire               Y_0 = (FUL_U1A == FUL_S1A); // Example14
wire               Y_1 = (FUL_U1A >= FUL_S1A); // Example15
```

In “Example12”, “FUL_S1A” is converted into unsigned before it is added to the unsigned variable “FUL_U1A”. In “Example13”, both “FUL_S1A” and “FUL_S1B” are signed so there is no signed-to-unsigned conversion. Notice that the expression type (signed or unsigned) is not affected by the LHS variable, as defined in Verilog-2005 Section 5.5.1. So even though “Y_U1C” is unsigned, the expression “FUL_S1A+FUL_S1B” is still signed. Both “Example12” and “Example13” are “bad” codes. “Example12” has signed-to-unsigned conversion in RHS expression, yielding incorrect result. “Example13” assigns signed RHS expression result to an unsigned LHS variable, which cannot represent negative values.

Signed-to-unsigned conversion rule is also applied to expressions that use relational and equality operators. “FUL_S1A” will be converted into unsigned in both “Example14” and “Example15”.

- ***Explicit signed variable is converted into unsigned when it is used in a concatenation.***

```
wire signed [3:0] S4b_A;
wire signed [3:0] S4b_B;
wire signed [4:0] S5b_Y = {S4b_A[3],S4b_A} + {S4b_B[3],S4b_B}; // Example16
```

In “Example16”, “S4b_A” and “S4b_B” are declared as signed variable. However, they are manually sign-extended in the addition expression and cause them to be converted into unsigned variable before the addition expression is evaluated.

- ***Explicit signed variable is converted into unsigned when it is used as its part-select or bit-select form.***

```
wire signed [3:0] S4b_A;
wire signed [3:0] S4b_B;
wire signed [4:0] S5b_X = S4b_A      + S4b_B     ;  // Example17
wire signed [4:0] S5b_Y = S4b_A[3:1] + S4b_B[3:1];  // Example18
wire signed [4:0] S5b_Z = S4b_A[3:0] + S4b_B[3:0];  // Example19
```

“Example17” yields a good result. “S4b_A” and “S4b_B” remain signed when the addition expression is evaluated. They are sign-extended correctly when the expression is evaluated.

“Example18” and “Example19” are bad implementations. “S4b_A” and “S4b_B” are both converted into unsigned variables before the addition is evaluated, causing incorrect results in “S5b_Y” and “S5b_Z”. Bit-select or part-select will convert the variable into unsigned, even if the part-select is the full vector.

- To illustrate the impact of signed-to-unsigned conversion, consider following examples:

```
wire             [2:0] U3b_A  =  3'b011; // +3
wire      signed [2:0] S3b_A  = 3'sb100; // -4
parameter signed [2:0] S3b_B  = 3'sb100; // -4
wire      signed [2:0] S3b_C  = 3'sb011; // +3
parameter signed [2:0] S3b_D  = 3'sb011; // +3

wire      signed [3:0] S4b_Y1 = U3b_A + S3b_A; // Example21 (Simulation: S4b_Y1 = +7)
```

```
wire      signed [3:0] S4b_Y2 = U3b_A + S3b_B; // Example22 (Simulation: S4b_Y2 = +7)
wire      signed [3:0] S4b_Y3 = S3b_C + S3b_B; // Example23 (Simulation: S4b_Y3 = -1)
parameter signed [3:0] S4b_Y4 = S3b_D + S3b_B; // Example24 (Simulation: S4b_Y4 = -1)
```

In “Example21”, “S3b_A” is converted into unsigned variable during evaluation of RHS expression. Instead of sign-extended, “S3b_A” is zero-extended into binary “0100”. RHS expression “U3b_A + S3b_A” is evaluated to a binary value “0111” (+7), which is clearly not what we want. On the other hand, “Example23” is a correct implementation. There is no signed-to-unsigned conversion during evaluation of RHS expression. “S3b_C” and “S3b_B” are sign-extended into binary values “0011” and “1100”, respectively. In simulation, “S4b_Y3” is binary “1111” (-1 in 2’s compliment), which is correct result. Please note that the signed-to-unsigned conversion rules are also applicable to parameters, as shown in “Example22” and “Example24”.

Signed-to-unsigned conversion is clearly a threat. However, Structural LINT tools can detect such problems. In Explicit Signed Variable Coding Style, signed-to-unsigned conversion is the first thing to be fixed. A simple rule-of-thumb is to make sure all operands, including variables, parameters, and constants, are declared (variable and parameter) or formatted (constant) as signed in a signed arithmetic expression.

C. Sign-casting input argument pitfall

(Reference: Verilog-2005 Section 5.5)

In Explicit Signed Variable Coding Style, system task $signed() is a useful syntax but it also comes with a pitfall. When a signed arithmetic expression must use an unsigned operand, sign-casting can be useful to turn it into signed. Verilog-2005 specified two system tasks for sign-casting, $signed() and $unsigned. System task $signed() turns its input argument into a signed operand. System task $unsigned() turns its input argument into an unsigned operand. Notice that these two system tasks don’t change the value and width of their input argument. Only the interpretations are changed. Practically speaking, system task $signed() is more useful than $unsigned() in the Explicit Signed Variable Coding Style so we will focus on $signed() in this section. Consider following example:

```
wire         [3:0] A_U4b;                                  // value range: 15 ~   0
wire  signed [3:0] B_S4b;                                  // value range:  7 ~  -8
wire  signed [5:0] X_S6b = A_U4b + B_S4b;                  // Example 25
wire  signed [5:0] Y_S6b = $signed(A_U4b) + B_S4b;         // Example 26
wire  signed [5:0] Z_S6b = $signed({1'b0,A_U4b}) + B_S4b; // Example 27
```

“Example25” mixes unsigned and signed variables in the expression, therefore “B_S4b” suffers from signed-to-unsigned conversion. “Example26” tries to fix the signed-to-unsigned conversion by casting the unsigned variable “A_U4b” into a signed operand. However, the way it uses system task $signed() still causes design issue. If “A_U4b” is “12” (“1100” in binary), the operand “$signed(A_U4b)” will be interpreted as a signed value “-4” and this is clearly not what the designer wants. “Example27” is a correct implementation, where “A_U4b” is zero-extended and concatenated result “{1’b0,A_U4b}” is not interpreted as a signed number [3]. In other words, one would need a signed 5-bit operand to hold a 4-bit unsigned value range.

D. Signed constant pitfalls

(Reference: Verilog-2005 Section 3.5.1)

Designers should also be careful about usage of constants when trying to avoid signed-to-unsigned conversion. Consider following example:

```
wire signed [3:0] A_S4b;
wire signed [4:0] X_S5b = A_S4b + 4'd3;                    // Example28
wire signed [4:0] Y_S5b = (A_S4b >= 4'd7) ? 4'sd7 : A_S4b; // Example29
wire signed [4:0] Z_S5b = (A_S4b >= 7) ? 4'sd7 : A_S4b;    // Example30
```

In both “Example28” and “Example29”, “A_S4b” is converted into unsigned because “4’d3” and “4’d7” are “based constant” so they are unsigned [1]. In “Example30”, constant “7” is a “simple decimal number” and it doesn’t cause signed-to-unsigned conversion because simple decimal numbers are signed [1].

In the examples, we can see another constant format “4’sd7”, where “s” indicates that the constant is signed and “d” is the base of the constant. However, care must be taken when specifying signed constants. For example:

```
wire signed [3:0] X_S4b = 4'sd8; // Example31 (bad code)
wire signed [3:0] Y_S4b = 4'sd4; // Example32
```

In “Example31”, “4’sd8” is not positive 8. Its binary form is “1000” and it would be negative 8 if we interpret binary “1000” as a signed number. A 4-bit signed number can only represent “+7” to “-8”, therefore it cannot represent

"+8". Specifying constant "4'sd8" appears to be correct but is misleading. In "Example32", "4'sd4" is correct because what it tries to represent is within its valid value range.

E. Operator precedence of arithmetic shift

(Reference: Verilog-2005 Section 5.1.12 and 5.1.2)

Multiplication or division by power of 2 doesn't require actual multiplier or divider. They are often done by "variable shift". Shifts can be done implicitly by concatenation or explicitly by shift operators. Consider following examples:

```
wire        [3:0] A_S4b;                                  // Implicit signed variable
wire signed [3:0] B_S4b;                                  // Explicit signed variable
wire        [7:0] Y1_S8b = {{5{A_S4b[3]}},A_S4b[3:1]};    // Example33
wire signed [7:0] Y2_S8b = B_S4b >>> 1;                   // Example34
wire signed [7:0] Y3_S8b = B_S4b <<< 1;                   // Example35
wire signed [7:0] Y4_S8b = B_S4b >>  1;                   // Example36 (bad code)
wire signed [7:0] Y5_S8b = B_S4b <<  1;                   // Example37 (not recommended)
```

"Example33" is a typical way in Implicit Signed Variable Coding Style to implement multiplication/division by power of 2. No variable is declared as signed but designer's intention about variable's signedness can be implicitly recognized from the manual sign-extension code for A_S4b. The implementation is correct but Structure LINT and Formal LINT tools will have a hard time analyzing the code due to lack of explicit signedness information.

For Explicit Signed Variable Coding Style, we avoid concatenation to prevent signed-to-unsigned conversion. Instead, we can use arithmetic shift operators, "<<<" and ">>>", to implement multiplication and division by power of 2, respectively. Arithmetic right-shift ">>>" will sign-extend a signed variable and zero-extend an unsigned variable, yielding correct result in arithmetic logic. As a simple rule of thumb, logical right-shift should be avoided in signed arithmetic logics.

"Example34" and "Example35" are the right ways to do shifts in arithmetic logics. "Example36" yields incorrect result because it uses logical right-shift on an explicit signed variable. "Example37" yields correct result but is not recommended to simplify the guideline for designers. Please note that the right operand of shift operator is always unsigned and has no effect on the signedness of the result [1].

Designers must also pay attention to operator precedence when using arithmetic shifts. According to Verilog LRM, shift operators have lower precedence than "+", "-", "*" and "/". This could be a little counter intuitive when designers think of shifts as multiplication and division. Consider following example:

```
wire signed [3:0] A;
wire signed [3:0] B;
wire signed [7:0] Y1 = A +  B <<< 1 ; // (A + B)*2 , Example38
wire signed [7:0] Y2 = A + (B <<< 1); //  A + B *2 , Example39
wire signed [7:0] Y3 = A +  B * 2   ; //  A + B *2 , Example40
```

Our intention here is to implement "A + B*2" where multiplication has higher precedence. In "Example38", addition will be evaluated first because operator "+" has higher precedence than operator "<<<". "Example 39 and 40" match our design intention. Notice that we need parenthesis around "B<<<1" in "Example 2".

F. Expression width and interim result pitfall

(Reference: Verilog-2005 Section 5.4)

Designers should keep the following table in mind to get more insight about how the length of an expression is determined (partial table extracted from Verilog-2005 Section 5.4):

Expression	Bit length	Comments
i op j, where op is: + - * / % & | ^ ^~ ~^	max(L(i),L(j))	
i op j, where op is: === !== == != > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...,j}	L(i)+..+L(j)	All operands are self-determined

A simple rule of thumb is to find out the largest operand in both RHS and LHS of the assignment, with exceptions at shift and conditional operators. Consider following example:

```
wire signed [3:0] A4b;
wire signed [4:0] B5b;
wire signed [5:0] C6b;
wire signed [8:0] S9b;
wire signed [5:0] Y6b = (S9b>9'sd100) ? (A4b>>>2) : (B5b+C6b); // Example41
wire signed [6:0] Y7b = (S9b>9'sd100) ? (A4b>>>2) : (B5b+C6b); // Example42
```

In "Example 41" and "Example42", only the widths of underlined variables shall be considered when we try to evaluate whether the assignments have arithmetic overflow issue. According to the expression length definition table, the select expression "S9b>9'sd100" in the conditional operator and the constant "2" in the arithmetic shift operator do not affect our evaluation for arithmetic overflow. In "Example41", the largest operands are 6-bit ("Y6b" and "C6b"), so all sub-expressions are evaluated as 6-bit. No overflow when "A4b>>>2" is evaluated as 6-bit expression. However, expression "B5b+C6b" has an overflow issue because it is evaluated as 6-bit expression. In "Example42", the largest operands are 7-bit ("Y7b"), therefore all sub-expressions are evaluated as 7-bit. Expression "B5b+C6b" has no overflow issue because it is evaluated as 7-bit expression.

Due to the expression length evaluation rule in Verilog LRM, designers must be careful about an "interim result overflow" problem, also mentioned in the Verilog-2005 LRM:

Example43:

```
wire [3:0] A = 4'b1100;
wire [3:0] B = 4'b0011;
wire [3:0] C = 4'b1110;
                              // Simulation result:
wire [5:0] Y0 = {A,B} >>> 2; // Y0 = 6'b110000
wire [2:0] Y1 =  A    >>> 1; // Y1 = 3'b110
wire [3:0] Y2 = (B+C) >>> 1; // Y2 = 4'b0000
wire [4:0] Y3 = (B+C) >>> 1; // Y3 = 5'b01000
wire [3:0] Y4 = (B+C) /2   ; // Y4 = 4'b1000
wire [4:0] Y5 = (B+C) /2   ; // Y5 = 5'b01000
wire [3:0] Y6 =  B+C-A     ; // Y6 = 4'b0101
wire [4:0] Y7 =  B+C-A     ; // Y7 = 5'b00101
```

In "Example43", for "Y2", RHS expression's value range is 15~0 and LHS variable "Y2"'s value range is also 15~0. It appears that LHS variable is wide enough to hold the result from RHS. However, (B+C) is evaluated as a 4-bit sub-expression because the largest operand in the entire assignment is 4-bit. Overflow occurs when "(B+C)" is evaluated. "Y4" has no interim result overflow because the constant "2" is 32-bit and therefore "B+C" is also evaluated as 32-bit expression and has no overflow and the result is correct. "B+C" for "Y6" does not have interim overflow either because "B+C-A" is evaluated all together.

As a rule of thumb, we suggest designers to be careful about "interim result overflow" issue when the value range is reduced by operators such as ">>>", "/" and "-".

G. Signed variable in assignment and module instantiation

(Reference: Verilog-2005 Section 5.5.1 and 12.3.11)

Signedness of LHS variable doesn't affect signedness of RHS expression in an assignment. Examples:

```
wire        [2:0] U3b_0    = 3'b111;     // Unsigned 7
wire signed [2:0] S3b_0    = 3'sb111;    // Signed  -1
wire        [2:0] U01_3b_0 = 3'b100;     // Unsigned 4
wire        [2:0] U01_3b_1 = 3'b111;     // Unsigned 7

wire signed [3:0] S00_4b_0 = U3b_0;      // Example44.a  S00_4b_0 = binary 0111 (Zero-extended)
wire signed [3:0] S00_4b_1 = U3b_0[2:0]; // Example44.b  S00_4b_1 = binary 0111 (Zero-extended)
wire signed [3:0] S00_4b_2 = S3b_0;      // Example44.c  S00_4b_2 = binary 1111 (Sign-extended)
wire signed [3:0] S00_4b_3 = S3b_0[2:0]; // Example44.d  S00_4b_3 = binary 0111 (Zero-extended)

wire signed [3:0] S01_4b_0 = U01_3b_0 - U01_3b_1; // Example45.a  S01_4b_0 = binary  1101 = -3
wire        [4:0] U01_5b_0 = S01_4b_0;            // Example45.b  U01_5b_0 = binary 11101 = 29
```

In "Example44.a" and "Example44.b", RHS variables are unsigned and zero-extended in the assignment even though they are assigned to LHS signed variables. In "Example44.d", RHS variable is converted to unsigned due to part-select. Therefore, it is zero-extended at the assignment despite the signed variable at LHS. Similarly, in "Example45.a" and "Example45.b", LHS variable's signedness doesn't affect RHS expression's signedness.

Signedness doesn't cross through module instantiation boundary either. Care must be taken to make sure the signedness of the variable matches the signedness of the module port. According to Verilog-2005 Section 12.3.11, the

sign attribute does NOT cross hierarchy. Signedness of a module port is determined by the port declaration inside of that module. For example, an unsigned variable connected to a signed module port doesn't make the port unsigned.

VII. Tool evaluation for Explicit Signed Variable Coding Style

We've been working with EDA vendors to polish their Formal LINT tools to have more relevant and accurate analysis result for signed arithmetic overflow verification. Due to the limitations mentioned in

Limitation of Formal LINT in Implicit Signed Variable Coding Style, we chose to focus on Explicit Signed Variable Coding Style, which we believe is the foundation of the success of Formal LINT verification for Implicit Signed Variable Coding Style.

We've observed promising results in the Explicit Signed Variable Coding Style category. In this section, we provide a summary of our Formal LINT tool evaluation. Furthermore, some complementary Structural or Formal LINT rules are also evaluated. Those complementary rules can help designers to avoid the pitfalls found in Explicit Signed Variable Coding Style.

A. Formal LINT results

Check Items		**Required LINT type**	**Commercial Solutions**
Signed-to-unsigned conversion	Due to mixed signed and unsigned operands in equation	Structural	Covered
	Due to part-select	Structural	Work in progress
	Due to concatenation	Structural	Work in progress
Bad sign-casting		Formal	Work in progress
Bad signed constant		Structural	Work in progress
Operator precedence of arithmetic shift (Tools won't know designer's intention)		n/a	n/a
Interim result overflow		Formal	Work in progress
Arithmetic overflow (LHS variable is not wide enough to hold result from RHS equation)		Formal	Covered

We evaluated different Formal LINT tools and the table summarizes the overall results. Although some of the complementary checks are still under development, they are all feasible according to EDA vendors' feedback. The vendors are committed to providing comprehensive solutions.

VIII. Summary

This paper provides a comprehensive exploration of the arithmetic overflow issue, delving into its intricacies. In the initial segment, we categorize common arithmetic logic coding and design styles into three distinct categories. Subsequently, we undertake a rigorous analysis of the capabilities of current Structural LINT and Formal LINT tools in verifying both unsigned and signed arithmetic overflow issues within these three styles. We emphasize that the crux of signed arithmetic overflow analysis lies in effectively conveying the signedness of variables to the verification tools.

Our investigation reveals that Formal LINT exhibits significant promise in verifying the "Explicit Signed Variable Coding Style." Consequently, Section 6 of this paper places a focused lens on presenting detailed coding guidelines and potential pitfalls for this specific coding style. Formal LINT EDA vendors can use the information to create complementary checks to provide comprehensive solutions for signed arithmetic overflow verification.

To encapsulate our findings succinctly, the table below offers a summary of the advantages and disadvantages associated with each of the three coding styles. Formal LINT can properly analyze Explicit Signed Variable Coding Style. However, the coding style has many pitfalls and may not be very friendly to the designers. Implicit Signed Variable Coding Style, on the other hand, has no tricky pitfalls and some designers prefer the coding style to have full control and clarity about operands manipulation. Formal LINT tools are still evolving and exploring the possibilities to accurately analyze Implicit Signed Variable Coding Style.

Drawing from the insights presented in this paper, we extend a recommendation to RTL designers. It is prudent for designers to proactively consider whether they intend to employ Formal LINT as their tool of choice for arithmetic overflow verification and subsequently determine the most suitable coding style to adopt. This proactive decision-making process can significantly enhance the efficiency and accuracy of the verification process.

Category	Type	Pitfall	Formal LINT limitation for Arithmetic Overflow Verification
Unsigned Arithmetic Logic	Design Type	None	No limitation
Implicit Signed Variable Coding Style	Coding Style	None	Lack of signedness information. Formal LINT currently cannot accurately analyze it. (Work-in-progress for EDA vendors)
Explicit Signed Variable Coding Style	Coding Style	Many	No showstopper but need complementary checks for designers to avoid pitfalls. (Work-in-progress for EDA vendors)

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