Proposal for Improving Expression Size Determination in IEEE 1800 SystemVerilog

Authors

This document contains a proposal for inclusion in the current SystemVerilog Language Reference Manual (LRM), IEEE 1800-2023, to clarify and formalize the rules for determining expression bit-widths.

Paragraphs with a gray border on the left are extracts taken directly from the latest version of the LRM. The sections containing text highlighted in red represent the specific content we believe could be replaced with our proposal. Our propositions, in green, are organized as follows:

Sections 11.6.1 to 11.6.4 (Expression bit-widths): This material introduces a revised formalization for bit-width determination, complete with rules governing this process, applicable to all SystemVerilog expressions.

Section 11.6.5 (Examples of bit-widths determination): This section provides examples demonstrating the use of the bit-width determination rules defined in the preceding section. The current version of the LRM offers a limited number of examples, primarily focusing on self-determined expressions. This draft proposes an update to this section, incorporating the new bit-width determination mechanism and extensive examples for all kinds of SystemVerilog expressions.

Section A (Algorithmic Overview): This content is proposed as a technical Appendix to the LRM. It formally describes the algorithms used to compute SystemVerilog expression bit-widths, which are only implicitly described in the current LRM version. This section also includes implementation considerations related to the efficient execution of the algorithms.

11.4.4 Relational operators

[...]
When one or both operands of a relational expression are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.
When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand. See 11.8.2 for more information.
If either operand is a real operand, then the other operand shall be converted to an equivalent real value and the expression shall be interpreted as a comparison between real values.
[...] [1, §11.4.4, p. 278]

11.4.5 Equality operators

[...]
When one or both operands are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.
When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand. See 11.8.2 for more information.
If either operand is a real operand, then the other operand shall be converted to an equivalent real value, and the expression shall be interpreted as a comparison between real values.
[...] [1, §11.4.5, p. 279]

11.4.8 Bitwise operators

[…]
For the binary bitwise operators, if one or both operands are unsigned, the result is unsigned. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.
If both operands are signed, the result is signed. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand. See 11.8.2 for more information.
For the unary bitwise negation operator, if the operand is unsigned, the result is unsigned. If the operand is signed, the result is signed.
[…] [1, §11.4.8, p. 281]

11.6.1 Rules for expression bit lengths

The rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.
The number of bits of an expression (known as the *size* of the expression) shall be determined by the operands involved in the expression and the context in which the expression is given.
A *self-determined expression* is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.
A *context-determined expression* is one where the bit length of the expression is determined by the bit length of the expression and by the fact that it is part of another expression. For example, the bit size of the right-hand expression of an assignment depends on itself and the size of the left-hand side.
Table 11-21 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 11-21, `i`, `j`, and `k` represent expressions of an operand, and `L(i)` represents the bit length of the operand represented by `i`.

Table 11-21—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
Unsized constant number	At least 32 bits	
Sized constant number	As given	
`i op j`, where op is: `+, -, *, /, %, &, |, ^, ^~, ~^`	$\max(\mathrm{L}(i), \mathrm{L}(j))$	
`op i`, where op is: `+, -, ~, ++, --`	$\mathrm{L}(i)$	
`i op j`, where op is: `===, !==, ==?, !=?, ==, !=, >, >=, <, <=`	1 bit	Operands are sized to $\max(\mathrm{L}(i), \mathrm{L}(j))$
`i op j`, where op is: `&&, ||, ->, <->`	1 bit	All operands are self-determined
`op i`, where op is: `&, ~&, |, ~|, ^, ~^, ^~, !`	1 bit	All operands are self-determined
`i op j`, where op is: `>>, <<, **, >>>, <<<`	$\mathrm{L}(i)$	j is self-determined
`i ? j : k`	$\max(\mathrm{L}(j), \mathrm{L}(k))$	i is self-determined
`{i, ..., j}`	$\mathrm{L}(i) + \cdots + \mathrm{L}(j)$	All operands are self-determined
`{i {j, ..., k}}`	$i \times (\mathrm{L}(i) + \cdots + \mathrm{L}(j))$	All operands are self-determined

Multiplication may be performed without losing any overflow bits by assigning the result to something wide enough to hold it.

[1, §11.6.1, p. 299]

The number of bits of any expression is determined by the operands and the context in which it occurs. Casting can be used to set the target width of an intermediate value (see 6.24).

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. The following typing system provides precise rules for determining expression bit widths in all situations.

The bit width of expressions is defined using the fundamental concepts:

Self-determined width The *self-determined width* is the intrinsic width of an expression: i.e. it is solely based on the

expression's internal structure and operands.

Resizing Expressions *may be resized* to bit-widths greater than or equal to their *self-determined width*. This operation may change the width of the internal expression.

11.6.2 Example of expression bit-length problem

[...] [1, §11.6.2, p. 300]

11.6.3 Expression categories for resizing

SystemVerilog expressions shall be categorized into two types based on their resizing behavior:

11.6.3.1 Atomically resizable expressions

Atomically resizable expressions *may be resized* without affecting their internal operand width. The following expressions are atomically resizable:

- Operands as defined in 11.2 (nets, variables, literals, function calls, etc.)
- Comparison expressions: `===`, `!==`, `==?`, `!=?`, `==`, `!=`, `>`, `>=`, `<`, `<=`
- Logical expressions: `&&`, `||`, `->`, `<->`
- Reduction expressions: `&`, `~&`, `|`, `~|`, `^`, `~^`, `^~`, `!`
- Assignment expressions: `=`
- Concatenation expressions: `{...}`
- Replication expressions: `{.{...}}`
- Set membership expressions: `inside`

When an atomically resizable expression is resized to a target width, only the expression's result shall be extended — its operands shall remain unmodified.

Rule (Atomic-Resize): If e has a *self-determined width* of t and n is larger than t and e is atomically resizable, then e *may be resized* to n.

11.6.3.2 Non-atomically resizable expressions

Non-atomically resizable expressions propagate resizing to their operands when a target width is specified. These expressions require their operands to be adjusted to specific widths based on the resizing rules. The following expression are not atomically resizable:

- Binary and bitwise expression: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `^~`, `~^`
- Unary arithmetic, bitwise, increment and decrement expressions: `+`, `-`, `~`, `++`, `--`
- Shift and power expression: `>>`, `<<`, `**`, `>>>`, `<<<`
- Conditional expression: `?:`

Binary arithmetic and bitwise expressions propagate the target width to both operands:

Rule (Binary-Resize): If a *may be resized* to n and b *may be resized* to n, then $a \oplus b$ *may be resized* to n.

Unary arithmetic, unary bitwise negation and unary increment and decrement expressions propagate the target width to their single operand:

Rule (Unary-Resize): If e *may be resized* to n, then $\oplus e$ *may be resized* to n.

Shift and power expressions propagate the target width only to the left operand, while the right operand remains *self-determined*:

Rule (Shift-Resize): If a *may be resized* to n and b has a *self-determined width* of t_b, then $a \oplus b$ *may be resized* to n.

Conditional expressions propagate the target width to both branch expressions, while the condition remains *self-determined*:

Rule (Conditional-Resize): If c has a *self-determined width* and t_e *may be resized* to n and f_e *may be resized* to n, then c `?` t_e `:` f_e *may be resized* to n.

11.6.4 *Self-determined* expression sizing rules

The *self-determined width* of an expression, solely based on its internal structure and operands, shall be computed according to the following rules:

11.6.4.1 Operands

For operands as defined in 11.2, the *self-determined width* is always well-defined and determined by their declaration, literal specification, or result type:

Rule (Operand-Size): If e is an operand and s its width, then e shall have a *self-determined width* of s.

Examples:

- Sized integer literals: `8'hFF` has a *self-determined width* of 8, `32'd123` has a *self-determined width* of 32,
- Unsized integer literals: `123`, `'hABC` have a *self-determined width* of at least 32 bits,
- Parameters, nets, variables and structure fields have their width defined by their declaration: `logic [15:0] data` has a *self-determined width* of 16,
- Bit-select: `data[5]` has a *self-determined width* of 1,
- Part-select: `data[7:0]` has a *self-determined width* of 8, `data[base +: 4]` has a *self-determined width* of 4,
- Function calls: Have their width defined by their return type — a function returning `logic [31:0]` has a *self-determined width* of 32,
- Variadic sized function calls: For functions whose return type depends on their arguments, the arguments' widths shall be determined as if they were in an assignment context. Once all argument widths are determined, the function's result type becomes known and defines the *self-determined width*.

11.6.4.2 Binary arithmetic and bitwise expressions

For binary arithmetic and bitwise expressions, the *self-determined width* is the maximum of the operand widths. The smaller operand is *resized* to match the larger operand's widths.

Rule (Binary-Left-Width): If a has a *self-determined width* of t and b *may be resized* to t, then $a \oplus b$ shall have a *self-determined width* of t.

Rule (Binary-Right-Width): If b has a *self-determined width* of t and a *may be resized* to t, then $a \oplus b$ shall have a *self-determined width* of t.

11.6.4.3 Unary expressions

For unary expressions (Unary arithmetic, unary bitwise negation and unary increment and decrement), the *self-determined width* is identical to the operand width.

Rule (Unary-Width): If e has a *self-determined width* of t, then $\oplus e$ shall have a *self-determined width* of t.

11.6.4.4 Relational and equality expressions

For relational and equality expressions, the *self-determined width* is always 1 bit. The smaller operand shall be resized to match the larger operand's width for comparison purposes.

Rule (Relational-Left-Width): If a has a *self-determined width* of t and b *may be resized* to t, then $a \oplus b$ shall have a *self-determined width* of 1.

Rule (Relational-Right-Width): If b has a *self-determined width* of t and a *may be resized* to t, then $a \oplus b$ shall have a *self-determined width* of 1.

11.6.4.5 Logical expressions

For binary logical expressions, the *self-determined width* is always 1 bit. All operands are *self-determined.*

Rule (Logical-Width): If a has a *self-determined width* of t_a and b has a *self-determined width* of t_b, then $a \oplus b$ shall have a *self-determined width* of 1.

11.6.4.6 Reduction expressions

For reduction expressions, including `!`, the *self-determined width* is always 1 bit. The operand is *self-determined.*

Rule (Reduction-Width): If e has a *self-determined width* of t, then $\oplus e$ shall have a *self-determined width* of 1.

11.6.4.7 Shift and power expressions

For shift and power expressions, the *self-determined width* is determined by the left operand. The right operand shall be *self-determined.*

Rule (Shift-Width): If a has a *self-determined width* of t and b has a *self-determined width* of t_b, then $a \oplus b$ shall have a *self-determined width* of t.

11.6.4.8 Assignment expressions

For assignment expressions, the *self-determined width* is determined by the left-hand side. When the left-hand side has a larger width than the right-hand side, the right-hand side shall *be resized.* Otherwise, the right-hand side shall be *self-determined.*

Rule (Assignment-Left-Width): If the left-hand side l has a width of t and e *may be resized* to t, then $l \oplus e$ shall have a *self-determined width* of t.

Rule (Assignment-Right-Width): If the left-hand side l has a width of t, e has a *self-determined width* of t_e and t is smaller than t_e, then $l \oplus e$ shall have a *self-determined width* of t.

11.6.4.9 Conditional expressions

For conditional expressions using the `?:` operator, the *self-determined width* is the maximum width of the two branch expressions. The smaller branch shall be resized to match the larger branch. The condition shall be *self-determined.*

Rule (Conditional-Left-Width): If c has a *self-determined width* of t_c, a has a *self-determined width* of t, and b *may be resized* to t, then $c?a:b$ shall have a *self-determined width* of t.

Rule (Conditional-Right-Width): If c has a *self-determined width* of t_c, b has a *self-determined width* of t, and a *may be resized* to t, then $c?a:b$ shall have a *self-determined width* of t.

11.6.4.10 Concatenation expressions

For concatenation expressions, the *self-determined width* is the sum of the *self-determined widths* of all operands.

Rule (Concatenation-Width): If e_1 has a *self-determined width* of t_1, ..., e_k has a *self-determined width* of t_k, and t is the sum of $t_1, \ldots, t_k$, then $\{e_1, \ldots, e_k\}$ shall have a *self-determined width* of t.

11.6.4.11 Replication expressions

The *self-determined width* of a replication is the *self-determined width* of the inner concatenation multiplied by the replication amount.

Rule (Replication-Width): If i is the amount of the replication and e_{in} has a *self-determined width* of t_{in}, and t is $i \times t_{in}$, then $\{i\{e_{in}\}\}$ shall have a *self-determined width* of t.

11.6.5 Example of self-determined expressions

```
logic [3:0] a;
logic [5:0] b;
logic [15:0] c;
initial begin
   a = 4'hF;
   b = 6'hA;
   $display("a*b=%h", a*b);   // expression size is self-determined
   c = {a**b};                // expression a**b is self-determined
                              // due to concatenation operator {}
   $display("a**b=%h", c);
   c = a**b;                  // expression size is determined by c
   $display("c=%h", c);
end
```

Simulator output for this example:

```
a*b=16   // 'h96 was truncated to 'h16 since expression size is 6
a**b=1   // expression size is 4 bits (size of a)
c=ac61   // expression size is 16 bits (size of c)
```

[1, §11.6.3, p. 301]

This section illustrates the application of sizing and type-derivation rules as defined in this specification. The examples are derived from the following declarations:

```
logic [7:0]  var8;      // 8-bit variable
logic [31:0] var32;     // 32-bit variable
logic [15:0] var16;     // 16-bit variable
logic        cond;      // condition signal
logic [63:0] result;    // 64-bit result variable
```

11.6.5.1 Basic Expression Sizing

This subsection demonstrates basic operand and binary expression sizing behavior.

Given the above declarations, the expression `var8` has *self-determined width* 8. By application of rule **Operand-Width**:

- `var8` is an operand with width 8, as declared in `logic [7:0] var8`.

The expression `var16[15:8] + 4'b1001` has *self-determined width* 8. By application of rule **Binary-Left-Width**:

- `var16[15:8]` has *self-determined width* 8 (by **Operand-Width**, part-select of 8 bits).
- `4'b1001` *may be resized* to 8 by rule **Resize**:
  - `4'b1001` has *self-determined width* 4 (by **Operand-Width**, sized literal).
  - 8 is greater than 4.
  - `4'b1001` is atomically resizable.

Application of rule **Binary-Right-Width** to this expression would not succeed, as it would require resizing `var16[15:8]` to 4 bits.

The expression `var16[5] + 8'hFF` has *self-determined width* 8. By rule **Binary-Right-Width**:

- `8'hFF` has *self-determined width* 8 (by **Operand-Width**, sized literal).
- `var16[5]` *may be resized* to 8 by rule **Resize**:
  - `var16[5]` has *self-determined width* 1 (by **Operand-Width**, bit-select).
  - 8 is greater than 1.
  - `var16[5]` is atomically resizable.

11.6.5.2 Relational Expression Example

This subsection illustrates the determination of sizing for relational expressions.

For the expression `var16 > 16'd100`, the resulting *self-determined width* is 1. By rule **Relational-Left-Width**:

- `var16` has *self-determined width* 16 (by **Operand-Width**, declaration `logic [15:0] var16`).
- `16'd100` *may be resized* to 16 (already 16 bits):
  - `16'd100` has *self-determined width* 16 (by **Operand-Width**, sized literal).

11.6.5.3 Reduction Expression Example

This subsection demonstrates sizing for reduction operations.

For the expression `&var16[7:0]`, the resulting *self-determined width* is 1. By rule **Reduction-Width**:

- `var16[7:0]` has *self-determined width* 8 (by **Operand-Width**, part-select of 8 bits).

11.6.5.4 Replication Expression Example

This subsection illustrates the effect of replication on expression width.

For the expression `{4{var8}}`, the resulting *self-determined width* is 32. By rule **Replication-Width**:

- The replication count i is 4.
- `var8` has *self-determined width* 8 (by **Operand-Width**, declaration `logic [7:0] var8`).
- The resulting width is $4 \times 8 = 32$.

11.6.5.5 Complex Replication with Concatenation

This subsection shows replication applied to a concatenated operand.

For the expression `{2{var16[7:0], 4'hF}}`, the resulting *self-determined width* is 24. By rule **Replication-Width**:

- The replication count i is 2.
- The inner concatenation `{var16[7:0], 4'hF}` has *self-determined width* 12 by rule **Concatenation-Width**:
  - `var16[7:0]` has *self-determined width* 8 (by **Operand-Width**, part-select).
  - `4'hF` has *self-determined width* 4 (by **Operand-Width**, sized literal).
  - Sum is $8 + 4 = 12$.
- The resulting width is $2 \times 12 = 24$.

11.6.5.6 Assignment with Target Width Extension

This subsection illustrates extension of a right-hand operand to match the assignment target.

For the expression `var32 = var16[7:0] + 1`, the resulting *self-determined width* is 32. By rule **Assignment-Left-Width**:

- Left-hand side `var32` has width 32 (by declaration `logic [31:0] var32`).
- Right-hand side `var16[7:0] + 1` *may be resized* to 32 by rule **Binary-Resize**:
  - `var16[7:0]` *may be resized* to 32 by rule **Resize**:
    - `var16[7:0]` has *self-determined width* 8 (by **Operand-Width**, part-select).
    - 32 is greater than 8.
    - `var16[7:0]` is atomically resizable.
  - `1` *may be resized* to 32 (unsized literals have at least 32 bits).

11.6.5.7 Assignment with Result Truncation

This subsection shows truncation of the right-hand result to match the left-hand target width.

For the expression `var8 = var32 + var16`, the resulting *self-determined width* is 8. By rule **Assignment-Right-Width**:

- Left-hand side `var8` has width 8 (by declaration `logic [7:0] var8`).
- Right-hand side `var32 + var16` has *self-determined width* 32 (by **Binary-Left-Width**):
  - `var32` has *self-determined width* 32 (by **Operand-Width**, declaration `logic [31:0] var32`).
  - `var16` *may be resized* to 32 by rule **Resize**:
    - `var16` has *self-determined width* 16 (by **Operand-Width**, declaration `logic [15:0] var16`).
    - 32 is greater than 16.
    - `var16` is atomically resizable.
- 8 is smaller than 32.

11.6.5.8 Conditional Expression with True Branch Determining Size

This subsection demonstrates conditional sizing where the true branch determines the resulting width.

For the expression `cond ? var32 : var8`, the resulting *self-determined width* is 32. By rule **Conditional-Left-Width**:

- Condition `cond` has *self-determined width* 1 (by **Operand-Width**, declaration `logic cond`).
- True branch `var32` has *self-determined width* 32 (by **Operand-Width**, declaration `logic [31:0] var32`).
- False branch `var8` *may be resized* to 32 by rule **Resize**:
  - `var8` has *self-determined width* 8 (by **Operand-Width**, declaration `logic [7:0] var8`).
  - 32 is greater than 8.
  - `var8` is atomically resizable.

11.6.5.9 Conditional Expression with False Branch Determining Size

This subsection demonstrates conditional sizing where the false branch determines the resulting width.

For the expression `cond ? var8 : var32`, the resulting *self-determined width* is 32. By rule **Conditional-Right-Width**:

- Condition `cond` has *self-determined width* 1 (by **Operand-Width**, declaration `logic cond`).
- False branch `var32` has *self-determined width* 32 (by **Operand-Width**, declaration `logic [31:0] var32`).
- True branch `var8` *may be resized* to 32 by rule **Resize**:
  - `var8` has *self-determined width* 8 (by **Operand-Width**, declaration `logic [7:0] var8`).
  - 32 is greater than 8.
  - `var8` is atomically resizable.

11.6.5.10 Conditional Expression with Context-Driven Sizing

This subsection demonstrates conditional sizing determined by the context of an assignment target.

For the expression `result = cond ? var32[7:0] : var32[15:8]`, the resulting *self-determined width* is 64. By rule **Assignment-Left-Width**:

- Left-hand side `result` has width 64 (by declaration `logic [63:0] result`).
- Right-hand side `cond ? var32[7:0] : var32[15:8]` *may be resized* to 64 by rule **Conditional-Resize**:
  - Condition `cond` has *self-determined width* 1 (by **Operand-Width**, declaration `logic cond`).
  - True branch `var32[7:0]` *may be resized* to 64 by **Resize**:
    - `var32[7:0]` has *self-determined width* 8 (by **Operand-Width**, part-select).
    - 64 is greater than 8.

    - `var32[7:0]` is atomically resizable.
  - False branch `var32[15:8]` *may be resized* to 64 by **Resize**:
    - `var32[15:8]` has *self-determined width* 8 (by **Operand-Width**, part-select).
    - 64 is greater than 8.
    - `var32[15:8]` is atomically resizable.

11.7 Signed expressions

[...] [1, §11.7, p. 301]

11.8.2 Steps for evaluating an expression

The following are the steps for evaluating an expression:

- Determine the expression size based upon the standard rules of expression size determination (see 11.6).
- Determine the sign of the expression using the rules outlined in 11.8.1.
- Propagate the type and size of the expression (or self-determined subexpression) back down to the context-determined operands of the expression. In general, any context-determined operand of an operator shall be the same type and size as the result of the operator. However, there are two exceptions:
  - If the result type of the operator is real and if it has a context-determined operand that is not real, that operand shall be treated as if it were self-determined and then converted to real just before the operator is applied.
  - The relational and equality operators have operands that are neither fully self-determined nor fully context-determined. The operands shall affect each other as if they were context-determined operands with a result type and size (maximum of the two operand sizes) determined from them. However, the actual result type shall always be 1 bit unsigned. The type and size of the operand shall be independent of the rest of the expression and vice versa.
- When propagation reaches a simple operand as defined in 11.5, then that operand shall be converted to the propagated type and size. If the operand shall be extended, then it shall be sign-extended only if the propagated type is signed.

[1, §11.8.2, p. 302]

11.8.3 Steps for evaluating an assignment

The following are the steps for evaluating an assignment:

- Determine the size of the right-hand side by the standard assignment size determination rules (see 11.6).
- If needed, extend the size of the right-hand side, performing sign extension if, and only if, the type of the right-hand side is signed.

[1, §11.8.3, p. 303]

11.8.4 Handling x and z in signed expressions

[...] [1, §11.8.4, p. 303]

A Technical Appendix: Algorithm Overview

This appendix presents an algorithm to compute the width of all sub-expressions of a SystemVerilog expression. The algorithm operates in two phases:

First, the *self-determined width* of the expression is computed using the algorithm 1. This algorithm traverses the expression tree bottom-up to determine the natural width of each expression based solely on its internal structure and operands.

Second, the expression and all its sub-expressions are resized to the target width using the algorithm 2. During this

propagation phase, all self-determined sub-expressions are resized to their *self-determined width*, while the other sub-expressions inherit their width from the surrounding context.

Assuming that call to the DETERMINE function are cached, the algorithm runs in linear time with respect to the number of operations in the SystemVerilog expression. The reasoning implemented in this algorithm follows the typing rules explained in the previous section 11.6.1.

Algorithm 1: Determine

Input: A SystemVerilog expression expr
Output: The self-determined width of expr

```
switch expr do
    when expr is an operand do
        return Γ(expr)
    when expr is lhs ⊕ rhs do                          // ⊕ can be +, -, *, /, %, &, |, ^, ^~, ~^
        lhs_w ← DETERMINE(lhs)
        rhs_w ← DETERMINE(rhs)
        return max(lhs_w, rhs_w)
    when expr is ⊕arg do                                // ⊕ can be +, -, ~, ++, --
        arg_w ← DETERMINE(arg)
        return arg_w
    when expr is lhs ⊕ rhs do                  // ⊕ can be ===, !==, ==?, !=?, ==, !=, >, >=, <, <=
        return 1
    when expr is lhs ⊕ rhs do                           // ⊕ can be &&, ||, ->, <->
        return 1
    when expr is ⊕arg do                          // ⊕ can be &, ~&, |, ~|, ^, ~^, ^~, !
        return 1
    when expr is lhs ⊕ rhs do                           // ⊕ can be >>, <<, **, >>>, <<<
        lhs_w ← DETERMINE(lhs)
        return lhs_w
    when expr is lval = rhs do
        lval_w ← φ(lval)
        return lval_w
    when expr is cond ? lhs : rhs do
        lhs_w ← DETERMINE(lhs)
        rhs_w ← DETERMINE(rhs)
        return max(lhs_w, rhs_w)
    when expr is {expr_1, ..., expr_N} do
        for i ∈ {1, ..., N} do
            width_i ← DETERMINE(expr_i)
        return Σ_{i=0}^{N} width_i
    when expr is {n arg} do
        arg_w ← DETERMINE(arg)
        return n × arg_w
```

References

[1] IEEE: Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2023 (Feb 2024). https://doi.org/10.1109/IEEESTD.2024.10458102

Algorithm 2: Propagate

Input: A SystemVerilog expression expr, A targetWidth to resize expr to.
Result: All sub-expressions of expr are annotated with their final width

switch expr **do**
  when expr *is an operand* **do**
    Annotate expr with targetWidth
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be +, -, *, /, %, &, |, ^, ^~, ~^
    PROPAGATE(lhs, targetWidth)
    PROPAGATE(rhs, targetWidth)
    Annotate expr with targetWidth
  when expr *is* $\oplus$arg **do** // $\oplus$ can be +, -, ~, ++, --
    PROPAGATE(arg, targetWidth)
    Annotate expr with targetWidth
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be ===, !==, ==?, !=?, ==, !=, >, >=, <, <=
    $\text{arg}_w \leftarrow \max(\text{DETERMINE}(\text{lhs}), \text{DETERMINE}(\text{rhs}))$
    PROPAGATE(lhs, arg_w)
    PROPAGATE(rhs, arg_w)
    Annotate expr with targetWidth
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be &&, ||, ->, <->
    PROPAGATE(lhs, DETERMINE(lhs))
    PROPAGATE(rhs, DETERMINE(rhs))
    Annotate expr with targetWidth
  when expr *is* $\oplus$arg **do** // $\oplus$ can be &, ~&, |, ~|, ^, ~^, ^~, !
    PROPAGATE(arg, DETERMINE(arg))
    Annotate expr with targetWidth
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be >>, <<, **, >>>, <<<
    PROPAGATE(lhs, targetWidth)
    PROPAGATE(rhs, DETERMINE(rhs))
    Annotate expr with targetWidth
  when expr *is* lval = rhs **do**
    PROPAGATE(rhs, $\max(\phi(\text{lval}), \text{DETERMINE}(\text{rhs}))$)
    Annotate expr with targetWidth
  when expr *is* cond ? lhs : rhs **do**
    PROPAGATE(cond, DETERMINE(cond))
    PROPAGATE(lhs, targetWidth)
    PROPAGATE(rhs, targetWidth)
    Annotate expr with targetWidth
  when expr *is* {expr_1, ... , expr_N} **do**
    for $i \in \{1, \ldots, N\}$ **do**
      PROPAGATE(expr_i, DETERMINE(expr_i))
    Annotate expr with targetWidth
  when expr *is* {n arg} **do**
    PROPAGATE(arg, DETERMINE(arg))
    Annotate expr with targetWidth