

# 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. It is designed to replace the content highlighted in the document “Highlighted LRM Sections for Revision.”

**Sections 11.6.1 to 11.6.4 (Expression bit-widths):** This material is proposed as replacement for the sub-sections in 11.6 of the current LRM. It introduces a revised formalization for bit-width determination, complete with rules governing this process, applicable to all SystemVerilog expressions. This content is a draft intended to replace existing content within the designated LRM section.

**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. This content is proposed to extend the existing Section 11.6.3 of the LRM. The current version of the LRM offers a limited number of examples, primarily focusing on self-determined expressions. This draft proposes an update for this section, with 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.

---

## 11.6.1 Expression bit-widths

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** We call *self-determined width* the bit-width that is intrinsic to 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

*Original sub-section of the LRM [...]*

## 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* of  $t_c$  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  $t_1, \dots, e_k$  has a *self-determined width* of  $t_k$ , and  $t$  is the sum of  $t_1, \dots, t_k$ , then  $\{e_1, \dots, e_k\}$  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 Examples of bit-widths determination

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

---

```
1 logic [7:0] var8;      // 8-bit variable
2 logic [31:0] var32;    // 32-bit variable
3 logic [15:0] var16;    // 16-bit variable
4 logic      cond;       // condition signal
5 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.
- 

## 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  $\text{expr}$ **Output:** The self-determined width of  $\text{expr}$ 

```

1 switch  $\text{expr}$  do
2   when  $\text{expr}$  is an operand do
3     return  $\Gamma(\text{expr})$ 
4   when  $\text{expr}$  is  $\text{lhs} \oplus \text{rhs}$  do                                //  $\oplus$  can be +, -, *, /, %, &, |, ^, ^~, ~^
5      $\text{lhs}_w \leftarrow \text{DETERMINE}(\text{lhs})$ 
6      $\text{rhs}_w \leftarrow \text{DETERMINE}(\text{rhs})$ 
7     return  $\max(\text{lhs}_w, \text{rhs}_w)$ 
8   when  $\text{expr}$  is  $\oplus \text{arg}$  do                                    //  $\oplus$  can be +, -, ~, ++, -
9      $\text{arg}_w \leftarrow \text{DETERMINE}(\text{arg})$ 
10    return  $\text{arg}_w$ 
11  when  $\text{expr}$  is  $\text{lhs} \oplus \text{rhs}$  do                                //  $\oplus$  can be ==, !=, ==?, !=?, ==, !=, >, >=, <, <=
12    return 1
13  when  $\text{expr}$  is  $\text{lhs} \oplus \text{rhs}$  do                                //  $\oplus$  can be &&, ||, ->, <->
14    return 1
15  when  $\text{expr}$  is  $\oplus \text{arg}$  do                                    //  $\oplus$  can be &, ~&, |, ~|, ^, ~^, ^~, !
16    return 1
17  when  $\text{expr}$  is  $\text{lhs} \oplus \text{rhs}$  do                                //  $\oplus$  can be », <<, **, >>, <<<
18     $\text{lhs}_w \leftarrow \text{DETERMINE}(\text{lhs})$ 
19    return  $\text{lhs}_w$ 
20  when  $\text{expr}$  is  $\text{lval} = \text{rhs}$  do
21     $\text{lval}_w \leftarrow \phi(\text{lval})$ 
22    return  $\text{lval}_w$ 
23  when  $\text{expr}$  is  $\text{cond} ? \text{lhs} : \text{rhs}$  do
24     $\text{lhs}_w \leftarrow \text{DETERMINE}(\text{lhs})$ 
25     $\text{rhs}_w \leftarrow \text{DETERMINE}(\text{rhs})$ 
26    return  $\max(\text{lhs}_w, \text{rhs}_w)$ 
27  when  $\text{expr}$  is  $\{\text{expr}_1, \dots, \text{expr}_N\}$  do
28    for  $i \in \{1, \dots, N\}$  do
29       $\text{width}_i \leftarrow \text{DETERMINE}(\text{expr}_i)$ 
30    return  $\sum_{i=0}^N \text{width}_i$ 
31  when  $\text{expr}$  is  $\{n \text{ arg}\}$  do
32     $\text{arg}_w \leftarrow \text{DETERMINE}(\text{arg})$ 
33    return  $n \times \text{arg}_w$ 

```



**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

```

1 switch expr do
2   when expr is an operand do
3     └ Annotate expr with targetWidth
4   when expr is lhs  $\oplus$  rhs do                                //  $\oplus$  can be +, -, *, /, %, &, |, ^, ^~, ~^
5     └ PROPAGATE(lhs, targetWidth)
6     └ PROPAGATE(rhs, targetWidth)
7     └ Annotate expr with targetWidth
8   when expr is  $\oplus$ arg do                                    //  $\oplus$  can be +, -, ~, ++, -
9     └ PROPAGATE(arg, targetWidth)
10    └ Annotate expr with targetWidth
11  when expr is lhs  $\oplus$  rhs do                                //  $\oplus$  can be ==, !=, ==?, !=?, ==, !=, >, >=, <, <=
12    └  $\arg_w \leftarrow \max(\text{DETERMINE}(\text{lhs}), \text{DETERMINE}(\text{rhs}))$ 
13    └ PROPAGATE(lhs,  $\arg_w$ )
14    └ PROPAGATE(rhs,  $\arg_w$ )
15    └ Annotate expr with targetWidth
16  when expr is lhs  $\oplus$  rhs do                                //  $\oplus$  can be &&, ||, ->, <->
17    └ PROPAGATE(lhs, DETERMINE(lhs))
18    └ PROPAGATE(rhs, DETERMINE(rhs))
19    └ Annotate expr with targetWidth
20  when expr is  $\oplus$ arg do                                    //  $\oplus$  can be &, ~&, |, ~|, ^, ~^, ^~, !
21    └ PROPAGATE(arg, DETERMINE(arg))
22    └ Annotate expr with targetWidth
23  when expr is lhs  $\oplus$  rhs do                                //  $\oplus$  can be », <<, **, >>, <<<
24    └ PROPAGATE(lhs, targetWidth)
25    └ PROPAGATE(rhs, DETERMINE(rhs))
26    └ Annotate expr with targetWidth
27  when expr is lval = rhs do
28    └ PROPAGATE(rhs,  $\max(\phi(\text{lval}), \text{DETERMINE}(\text{rhs}))$ )
29    └ Annotate expr with targetWidth
30  when expr is cond ? lhs : rhs do
31    └ PROPAGATE(cond, DETERMINE(cond))
32    └ PROPAGATE(lhs, targetWidth)
33    └ PROPAGATE(rhs, targetWidth)
34    └ Annotate expr with targetWidth
35  when expr is {expr1, . . . , exprN} do
36    └ for  $i \in \{1, \dots, N\}$  do
37      └ PROPAGATE(expri, DETERMINE(expri))
38    └ Annotate expr with targetWidth
39  when expr is {n arg} do
40    └ PROPAGATE(arg, DETERMINE(arg))
41    └ Annotate expr with targetWidth

```