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.

1.1 Bidirectional typing operations

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.

1.2 Expression categories for resizing

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

1.2.1 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.

1.2.2 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* 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.

1.3 *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:

1.3.1 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*.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.3.5 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.

1.3.6 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.

1.3.7 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.

1.3.8 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.

1.3.9 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.

1.3.10 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.

1.3.11 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.

2 Examples

Consider the following SystemVerilog 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
```

```
logic cond;             // condition signal
logic [63:0] result;    // 64-bit result variable
```

2.1 Basic Expression Sizing

In the previous context, the expression `var8` has *self-determined width* 8. Indeed, by rule **Operand-Width**:

- `var8` is an operand with width 8 (by declaration `logic [7:0] var8`)

In the previous context, the expression `var16[15:8] + 4'b1001` has *self-determined width* 8. Indeed, by rule **Binary-Left-Width**:

- `var16[15:8]` has *self-determined width* 8 (by rule **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 rule **Operand-Width**, sized integer literal)
  - 8 is larger than 4
  - `4'b1001` is atomically resizable (operands are atomically resizable)

If we tried to apply rule **Binary-Right-Width** on the previous expression we would end up stuck resizing `var16[15:8]` to 4 bits.

In the previous context, the expression `var16[5] + 8'hFF` has *self-determined width* 8. Indeed, by rule **Binary-Right-Width**:

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

2.2 Relational Expression Example

In the previous context, the expression `var16 > 16'd100` has *self-determined width* 1. Indeed, by rule **Relational-Left-Width**:

- `var16` has *self-determined width* 16 (by rule **Operand-Width**, declaration `logic [15:0] var16`)
- `16'd100` *may be resized* to 16 — this is actually not needed as it already has width 16:
  - `16'd100` has *self-determined width* 16 (by rule **Operand-Width**, sized integer literal)

2.3 Reduction Expression Example

In the previous context, the expression `&var16[7:0]` has *self-determined width* 1. Indeed, by rule **Reduction-Width**:

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

2.4 Replication Expression Example

In the previous context, the expression `{4{var8}}` has *self-determined width* 32. Indeed, by rule **Replication-Width**:

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

2.5 Complex Replication with Concatenation

In the previous context, the expression `{2{var16[7:0], 4'hF}}` has *self-determined width* 24. With rule **Replication-Width**:

- The replication amount 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 rule **Operand-Width**, part-select)
  - `4'hF` has *self-determined width* 4 (by rule **Operand-Width**, sized literal)
  - Sum is $8 + 4 = 12$
- The result width is $2 \times 12 = 24$

2.6 Assignment with Target Width Extension

In the previous context, the expression `var32 = var16[7:0] + 1` has *self-determined width* 32. With 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 rule **Operand-Width**, part-select)
    - 32 is larger than 8
    - `var16[7:0]` is atomically resizable (operands are atomically resizable)
  - `1` *may be resized* to 32 (unsized literals have *self-determined width* at least 32)

2.7 Assignment with Result Truncation

In the previous context, the expression `var8 = var32 + var16` has *self-determined width* 8. Indeed, according to the 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 rule **Binary-Left-Width**:
  - `var32` has *self-determined width* 32 (by rule **Operand-Width**, declaration `logic [31:0] var32`)
  - `var16` *may be resized* to 32 by rule **Resize**:
    - `var16` has *self-determined width* 16 (by rule **Operand-Width**, declaration `logic [15:0] var16`)
    - 32 is larger than 16
    - `var16` is atomically resizable (operands are atomically resizable)
- 8 is smaller than 32

2.8 Conditional Expression with True Branch Determining Size

In the previous context, the expression `cond ? var32 : var8` has *self-determined width* 32. Indeed, according to the rule **Conditional-Left-Width**:

- Condition `cond` has *self-determined width* 1 (by rule **Operand-Width**, declaration `logic cond`)
- True branch `var32` has *self-determined width* 32 (by rule **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 rule **Operand-Width**, declaration `logic [7:0] var8`)
  - 32 is larger than 8
  - `var8` is atomically resizable (operands are atomically resizable)

2.9 Conditional Expression with False Branch Determining Size

In the previous context, the expression `cond ? var8 : var32` has *self-determined width* 32. Indeed, according to the rule **Conditional-Right-Width**:

- Condition `cond` has *self-determined width* 1 (by rule **Operand-Width**, declaration `logic cond`)
- False branch `var32` has *self-determined width* 32 (by rule **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 rule **Operand-Width**, declaration `logic [7:0] var8`)
  - 32 is larger than 8
  - `var8` is atomically resizable (operands are atomically resizable)

2.10 Conditional Expression with Context-Driven Sizing

In the previous context, the expression `result = cond ? var32[7:0] : var32[15:8]` has *self-determined width* 64. Indeed, 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 rule **Operand-Width**, declaration `logic cond`)
  - True branch `var32[7:0]` *may be resized* to 64 by rule **Resize**:
    - `var32[7:0]` has *self-determined width* 8 (by rule **Operand-Width**, part-select)
    - 64 is larger than 8
    - `var32[7:0]` is atomically resizable (operands are atomically resizable)
  - False branch `var32[15:8]` *may be resized* to 64 by rule **Resize**:
    - `var32[15:8]` has *self-determined width* 8 (by rule **Operand-Width**, part-select)
    - 64 is larger than 8
    - `var32[15:8]` is atomically resizable (operands are atomically resizable)

A 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 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 $\oplus$ rhs **do** // $\oplus$ can be +, -, *, /, %, &, |, ^, ^~, ~^
    $\text{lhs}_\text{w} \leftarrow$ DETERMINE(lhs)
    $\text{rhs}_\text{w} \leftarrow$ DETERMINE(rhs)
    return $\max\left(\text{lhs}_\text{w}, \text{rhs}_\text{w}\right)$
  when expr *is* $\oplus$arg **do** // $\oplus$ can be +, -, ~, ++, -
    $\text{arg}_\text{w} \leftarrow$ DETERMINE(arg)
    return arg_w
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be ===, !==, ==?, !=?, ==, !=, >, >=, <, <=
    return 1
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be &&, ||, ->, <->
    return 1
  when expr *is* $\oplus$arg **do** // $\oplus$ can be &, ~&, |, ~|, ^, ~^, ^~, !
    return 1
  when expr *is* lhs $\oplus$ rhs **do** // $\oplus$ can be », <<, **, »>, <<<
    $\text{lhs}_\text{w} \leftarrow$ DETERMINE(lhs)
    return lhs_w
  when expr *is* lval = rhs **do**
    $\text{lval}_\text{w} \leftarrow \phi(\text{lval})$
    return lval_w
  when expr *is* cond ? lhs : rhs **do**
    $\text{lhs}_\text{w} \leftarrow$ DETERMINE(lhs)
    $\text{rhs}_\text{w} \leftarrow$ DETERMINE(rhs)
    return $\max\left(\text{lhs}_\text{w}, \text{rhs}_\text{w}\right)$
  when expr *is* {expr_1, ... , expr_N} **do**
    for $i \in \{1, \ldots, N\}$ **do**
      $\text{width}_\text{i} \leftarrow$ DETERMINE(expr_i)
    return $\sum_{i=0}^{N} \text{width}_\text{i}$
  when expr *is* {n arg} **do**
    $\text{arg}_\text{w} \leftarrow$ DETERMINE(arg)
    return $n \times \text{arg}_\text{w}$

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

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