# A mechanised bidirectional type system for bit-width inference in SystemVerilog

Gabriel Desfrene[1,2], Quentin Corradi[2]
Michalis Pardalos[2], and John Wickerson[2]

[1] École normale supérieure – PSL, Paris, France
gabriel.desfrene@ens.psl.eu
[2] Imperial College London, United Kingdom
{q.corradi22, michail.pardalos17, j.wickerson}@imperial.ac.uk



**Abstract** SystemVerilog remains one of the most widely used languages for designing and verifying digital circuits. Despite its importance, the SystemVerilog standard suffers from ambiguities and inconsistent implementations between tools, creating portability challenges. Formal methods can provide precise semantics to address these issues.
As a first step, we tackle SystemVerilog's mechanism for determining the bit-width of each expression that appears in a design. Determining the bit-width of SystemVerilog expressions is surprisingly subtle because an expression's bit-width is context-dependent: it can depend on both its children *and* its parents.
We present a comprehensive formal treatment of this problem. First, we develop a Rocq formalization of the SystemVerilog specification as currently written. We then construct a bidirectional type inference system that captures the context-dependent nature of SystemVerilog expressions and prove its equivalence to our formalization of the existing IEEE standard via a machine-checked proof. Finally, we provide a reference implementation that determines expression bit-widths in linear time and prove its correspondence to our type system, also machine-checked in Rocq.
Our approach provides a precise mathematical foundation for this aspect of the language while maintaining full compatibility with the existing standard. Additionally, we propose improvements to the SystemVerilog standard that eliminate several redundancies and ambiguities while preserving existing behaviour.

**Keywords:** Bidirectional typing · Type inference · Hardware description languages · Formal semantics · Context-dependent typing · Machine-checked proofs · Language standardization · SystemVerilog

## 1 Introduction

Verilog and its successor SystemVerilog have been foundational to digital design since 1984 [12]. These hardware description languages have enabled the design, verification, and synthesis of everything from simple logic gates to complex

System-on-Chip (SoC) designs. Today, they represent the industry standard for both FPGA and ASIC design flows, with SystemVerilog being adopted across the majority of functional verification projects according to recent industry surveys [13]. The language is now defined by IEEE 1800-2023, officially titled the "SystemVerilog Language Reference Manual" (LRM) [16].

Despite SystemVerilog's widespread adoption, the language specification presents significant challenges for both tool developers and users. The current standard spans 1,354 pages—nearly twice the length of the ISO C standard's 758 pages—and contains extensive prose descriptions that can lead to ambiguous interpretations. This complexity has resulted in what industry experts characterize as 'unclear specifications on how to interpret SystemVerilog code' [22]. The practical consequences of these specification ambiguities are substantial. As Dave Rich, Technical Chair of the IEEE SystemVerilog Working Group, observes:

> Many users avoid adopting SystemVerilog because feature support from different tools and vendors of the rapidly changing LRM had been so inconsistent. To this day, people continue using Verilog-1995 syntax and avoid using features added by Verilog-2001 (e.g., ANSI-style ports and the power operator) [22].

This inconsistency across tools has created a fragmented ecosystem where users cannot rely on portable behavior, leading to the development of extensive test suites like `sv-tests` to characterize tool-specific interpretations [3].

To address these specification ambiguities and ensure reliable tool implementations, researchers have increasingly turned to formal methods. These approaches aim to provide mathematically rigorous foundations for SystemVerilog tools, enabling developers and enterprises to build designs and verification environments that demonstrably conform to the LRM. Recent efforts include the Vera equivalence checker [20], the Lutsig verified Verilog compiler [17], and testing frameworks such as Verismith [6, 15] and ChiGen [24].

However, to achieve a fully verified SystemVerilog toolchain—one that provides end-to-end guarantees from source code to final implementation—a critical missing component is a formally verified SystemVerilog type checker. Previous formalization efforts have not addressed bit-width inference with sufficient rigor. Featherweight Verilog provides a minimal core calculus for Verilog's synthesizable subset but employs only generic types without bit-width inference [14]. Chen et al.'s tractable operational semantics for SystemVerilog handles bit-width computations algorithmically but lacks formal typing rules necessary for mechanized correctness proofs [2]. Choi et al.'s denotational semantics approach assumes bit-widths are predetermined [4] and Lööw's analysis of the LRM inconsistencies focuses on simulation semantics rather than bit-width determination rules [18].

This paper addresses this gap by developing the first bidirectional type system for SystemVerilog bit-width determination. Bidirectional typing, as demonstrated by Pierce and Turner, provides an elegant foundation for reasoning about subtyping [11, 21]. This is particularly relevant because bit-width determination in SystemVerilog can naturally be formulated as a subtyping problem: a bit-vector of width $s$ is a subtype of a bit-vector of width $t$ whenever $t \geqslant s$. Recognizing

this correspondence between bit-vector width inference and bidirectional typing underpins the theoretical foundation of our formal framework.

However, in SystemVerilog, the bit-width of a sub-expression cannot be determined in isolation; it depends on contextual information that may itself rely on bit-widths computed earlier. This circular dependency leads the LRM to specify a complex two-phase algorithm for bit-width determination. Our approach resolves this interdependency in a principled way while providing formal guarantees to the LRM algorithm. Furthermore, our formal framework is designed to facilitate integration with existing verification and compilation tools, potentially enhancing the formal guarantees provided by the previously cited systems.

We claim the following contributions:

**Contribution 1** We formalize an important subset of the IEEE 1800 specification for bit-width determination in the Rocq theorem prover [23], providing a precise mathematical interpretation of the standard's prose descriptions.

**Contribution 2** We develop a bidirectional type system that captures the context dependent nature of SystemVerilog expression bit-widths, introducing formal typing rules and proving key properties.

**Contribution 3** We provide machine-checked proofs that our type system is equivalent to our formalization of the LRM specification.

**Contribution 4** We derive a reference implementation that computes bit-widths in linear time and provide a Rocq-checked proof of its correctness with respect to our type system.

**Contribution 5** We present a proposal to improve the treatment of bit-width determination in the IEEE 1800 standard. This proposal is grounded in the formal framework developed in this work and aims to provide a clear, unambiguous definition of the bit-width of SystemVerilog expressions.

## 2 Formalizing the LRM

SystemVerilog is fundamentally designed to describe hardware circuits, which imposes specific constraints on expression evaluation. All operations must ultimately be translated into physical circuitry. Therefore, all operands within expressions must be appropriately sized according to the constraints imposed by both the operation type and the dimensions of participating operands. For instance, the LRM establishes that binary operations, such as addition, require both operands to have identical bit-width, which subsequently determines the width of the resulting expression. This bit-width requirement ensures that the hardware synthesis process can generate consistent and predictable circuit implementations. Crucially, this constraint guarantees proper handling of signed operations, where operand sign extension must be correctly managed to preserve arithmetic semantics.

In this section, we explain how SystemVerilog's expression bit-width mechanisms work. After presenting examples and a note on our expression grammar, we

detail our Rocq formalization approach for modeling this mechanism, based on the LRM requirements.

### 2.1 How Sizing Works in SystemVerilog

The *self-determined width* of an expression (including operands) is the bit-width determined solely by the expression itself, independent of its evaluation context. This constitutes an intrinsic property of the expression, corresponding to what the LRM refers to in its latest version [16, §I.11.6] as a *self-determined* expression.[3] Practically, this property ensures that operand widths can be computed through local analysis alone. For instance, the self-determined width of an 8-bit bus is 8, as is the self-determined width of the literal `8'd128`.

The computation of self-determined width follows a recursive, bottom-up traversal strategy that we will call determine. This process begins with leaf nodes (operands) whose bit-width are explicitly known, then propagates upward through the abstract syntax tree according to operator-specific sizing rules. Each node of the abstract syntax tree is annotated with its self-determined width.

In the following examples, we assume that the operand `x` is defined with a self-determined width of 8 bits, `y` with a self-determined width of 4 bits, and `z` with a self-determined width of 1 bit. These assumptions correspond to the following SystemVerilog declarations:

```
1 logic [7:0] x;  // 8-bit variable
2 logic [3:0] y;  // 4-bit variable
3 logic       z;  // 1-bit variable
4
5 assign ex1 = x + (y + z);
6 assign ex2 = {x, y + z};
```

The self-determined width represents the minimal bit-width that each sub-expression requires for proper evaluation. Figure 1 illustrates the determine process for two expressions. During this phase, binary operations return the maximum self-determined width of their arguments, thereby enforcing the bit-width uniformity constraint described above. Concatenation operations exhibit different behavior, as the LRM mandates that the self-determined width of such expressions equals the sum of the self-determined widths of all participating arguments.

Leaf nodes provide known self-determined widths, while operation nodes compute their widths based on their children's dimensions. The red arrows indicate bit-width information flow during the computation process. Nodes highlighted in orange represent the currently processed elements during traversal. Each AST node annotation is represented with a semicolon followed by its self-determined width.

[3] This concept is subsequently applied to sign and type computation in other sections of the LRM

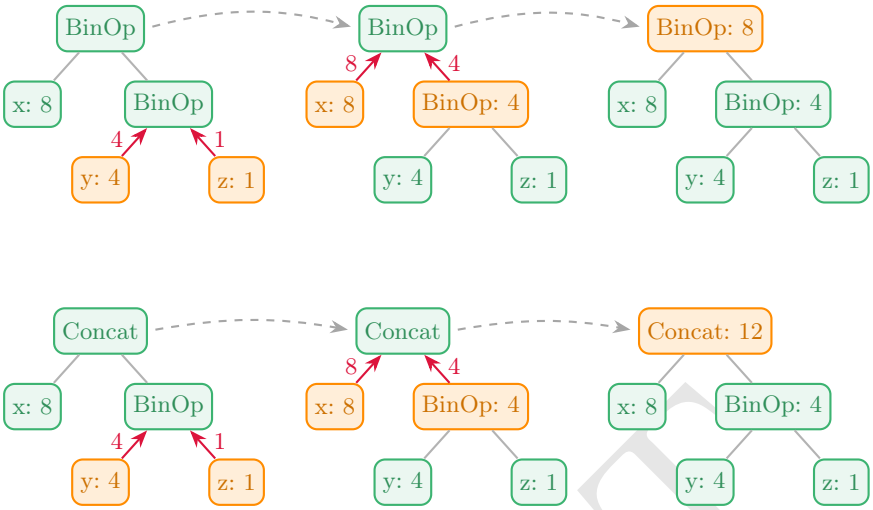


**Figure 1.** *Self-determined* width computation (determine phase)

As seen in the previous examples, the determine process only computes the self-determined width of the top level expression. The width of each sub-expression is computed during a second process called propagate. This process flows bit-width information back down the AST. During this process, sub-expressions are not always sized to their self-determined width; instead, they *may be resized* according to the surrounding context. We call this context-determined width the *final width*.

This second phase is mandatory, because evaluating expressions by only taking into account their self-determined width could result in an information loss. Consider an addition operation involving two operands: evaluating the result strictly according to the self-determined width would yield an output sized to match the maximum operand width, thereby discarding the carry-out bit.

An alternative solution could be to require that arithmetic operations always produce results that are one bit larger than their operands, in order to preserve carry information. However, this approach proves unsatisfactory in practice, as it would lead to systematic circuit growth throughout the design hierarchy, consuming unnecessary hardware resources when carry-out bits are not required.

SystemVerilog adopts a flexible strategy to address this sizing dilemma. When carry-bit preservation is necessary, designers can specify larger contexts for expression evaluation. The propagate phase ensures that sub-expressions are appropriately resized according to their contextual requirements, thereby generating results with the precise bit-widths demanded by the surrounding logic.

Figure 2 illustrates this process for a binary addition operation. The notation $x \rightarrow y$ denotes that a node possesses a self-determined width of $x$ bits and must be adjusted to a final width of $y$ bits to satisfy the contextual constraints imposed by the expression.

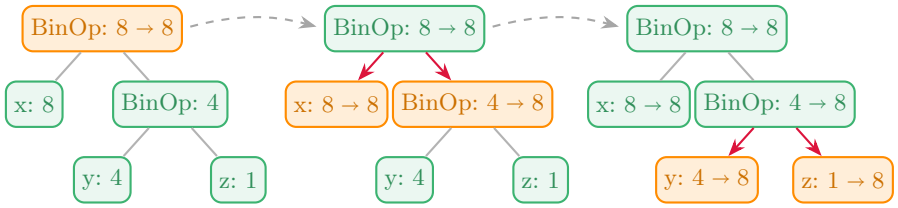


**Figure 2.** Context-dependent resizing (propagate phase) for binary addition

Certain operators, such as concatenation, establish context boundaries that isolate their arguments from external sizing influences. These boundaries ensure that specific operands remain self-determined regardless of the surrounding context. Figure 3 illustrates this concept, where context boundaries (depicted as arcs across edges) prevent bit-width information from propagating downwards beyond the operator. Consequently, the nested binary operation maintains its self-determined width of 4 bits rather than being resized to match the bit-width of the `x` variable.

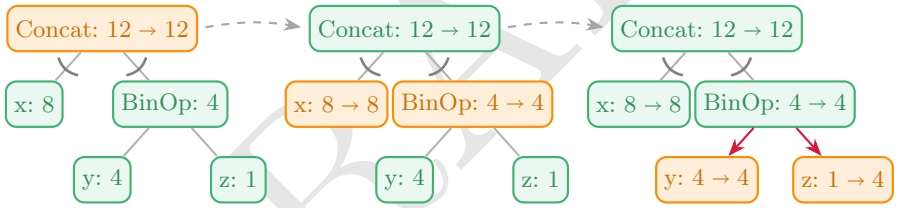


**Figure 3.** Context boundary enforcement in concatenation operations

In summary, the complete sizing mechanism operates through a two-phase process. The first phase, determine, *synthesizes* bit-width information throughout the expression hierarchy. This phase compute the final widths of the top-level and all self-determined sub-expressions. This synthesis process follows LRM constraints specific to each node type.

The second phase, propagate, computes the final width for all non-self-determined sub-expressions by propagating bit-width information downward through the AST. These sub-expressions derive their final width from the surrounding context and from their intrinsic properties.

This dual-phase approach ensures operations possess sufficient bit-width to compute expected results. Additionally, it maintains minimal hardware overhead through appropriate bit-width constraints (See theorem 1).

### 2.2 Formal Expression Model

For our Rocq formalization of the LRM mechanism, we present a simplified formalization of SystemVerilog expressions that captures all the typing behaviors. In SystemVerilog, expressions are defined as follows:

> An *expression* is a construct that combines *operands* with *operators* to produce a result that is a function of the values of the operands and the semantic meaning of the operator. [16, §I.11.2]

Operands represent atomic values in SystemVerilog that cannot be decomposed into further operations. This class encompasses literals, variables, structure members, union members, function calls, and other primitive constructs (the complete enumeration is in [16, §I.11.2]). For our formalization, we require operands to satisfy the following property:

*Property 1.* The self-determined width of an operand is always well-defined and determined by its declaration, literal specification, or result type.

We assume a normalization preprocessing step that ensures no operand contains nested expressions. This transformation introduces fresh variables for expressions composing operands, particularly targeting function calls. Since function calls are classified as operands, their self-determined width is defined by the function's return type. For variadic functions such as type casts, the output width can be determined from the function signature and the self-determined widths its arguments (which themselves require typing analysis). This normalization guarantees that all operands possess well-defined self-determined width while simplifying subsequent operations.

We categorize SystemVerilog operations by their sizing behaviors, yielding the expression grammar shown in figure 4. In this grammar, $o$ denotes a SystemVerilog operand, $l$ represents a left-hand side expression during assignment, and $n$ is an integer. We assume that all constant expressions have been pre-evaluated.

In SystemVerilog reduction operations apply a bitwise operator across all bits of one operand (e.g., `&x` is true if all bits of `x` are 1). Concatenation builds a wider bit vector by joining operands side by side (e.g., `{a, b}` places `a`'s bits above `b`'s). Replication repeats an expression a fixed number of times to form a larger vector (e.g., `{3{a}}` concatenates three copies of `a`).

### 2.3 Grammar Simplifications

We present the following observations regarding our grammar formalization:

**Assignments** Compound assignment operators (`+=`, etc.) are treated as syntactic sugar, e.g. `a += b` is interpreted as `a = a + b`, as specified in [16, §I.11.4.1].

**Replication Operator** The LRM constrains the `Replication` operator to operate solely on `Concatenation` expressions. We adopt a more permissive approach by allowing replication of any expression because it simplifies our typing rules.

| | | |
|---|---|---|
| $e ::= o$ | | operands |
| \| | `BinaryOp(`$e_1$`, `$e_2$`)` | binary operations (`+`, `-`, `*`, `/`, ...) |
| \| | `UnaryOp(`$e$`)` | unary operations (`+`, `-`, `~`, `++`, `--`) |
| \| | `ComparisonOp(`$e_1$`, `$e_2$`)` | comparison operations (`==`, `!=`, `>`, `>=`, ...) |
| \| | `LogicOp(`$e_1$`, `$e_2$`)` | logical operations (`&&`, `\|\|`, `->`, `<->`) |
| \| | `ReductionOp(`$e$`)` | reduction operations (`!`, `&`, `\|`, `^`, ...) |
| \| | `ShiftOp(`$e_1$`, `$e_2$`)` | shift operations and power (`>>`, `<<`, `**`, ...) |
| \| | `AssignmentOp(`$l$`, `$e$`)` | assignment |
| \| | `Conditional Op(`$e_1$`, `$e_2$`, `$e_3$`)` | conditional expression |
| \| | `Concatenation(`$e_1$`, ..., `$e_k$`)` | concatenation |
| \| | `Replication(`$n$`, `$e$`)` | replication |

**Figure 4.** SystemVerilog expression grammar categorized by typing behavior

**Inside Operator** We do not formalize the `Inside` operator since it can be derived from a sequence of `Or` operations during normalization. This aligns with the semantics prescribed in [16, §I.11.4.13].

**Streaming Operators** Streaming operators are excluded from our formalization as they behave identically to concatenation with respect to bit-width determination. The streaming operator's reordering of the bits does not affect its overall bit-width.

**Dist Operator** Similarly, we exclude the `dist` operator since Annex A of the LRM restricts its usage to constraint blocks, which fall outside the scope of general expression evaluation.

### 2.4 Formalizing the LRM

The LRM does not present a unified typing system for bit-width determination. Instead, it provides fragments of an algorithm distributed across multiple sections. This algorithm consists of two primary phases: determine and propagate, as described in §I.11.8.2:

> **Determine** the expression size based upon the standard rules of expression size determination [...]. **Propagate** the type and size of the expression (or *self-determined* subexpression) back down to the *context-determined* operands of the expression. [16, §I.11.8.2]

The LRM defines *context-determined* expressions as follows:

> 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. [16, §I.11.6.1]

*The Determine Phase.* The first phase is well-defined by Table 11-21 in [16, §I.11.6.1]. This phase computes the self-determined width for each expression in a bottom-up fashion, as seen in the example in figure 1. We formalize this as a recursive function determine : Expr → ℕ, defined in figure 5.

This formalization relies on two auxiliary functions. The function $\Gamma$ maps each operand in $\mathcal{O}$ (the set of all operands) to its declared bit-width, providing the foundation for all bit-width computations. This function is a direct consequence of property 1 that we established for operands. The function $\phi$ determines the bit-width of left-hand sides in assignments. This function exists because assignment targets have a simpler syntactic structure than general expressions (see [16, §A.8.5]). They consist of operands, concatenations of assignment targets, or streaming concatenations of assignment targets. This restricted structure allows us to construct a simple recursive function to compute their width.

$$
\begin{aligned}
\mathsf{determine}(o) &= \Gamma(o) \quad (\text{where } o \in \mathcal{O}) \\
\mathsf{determine}(\texttt{BinaryOp}(e_1, e_2)) &= \max(\mathsf{determine}(e_1), \mathsf{determine}(e_2)) \\
\mathsf{determine}(\texttt{UnaryOp}(e)) &= \mathsf{determine}(e) \\
\mathsf{determine}(\texttt{ComparisonOp}(e_1, e_2)) &= 1 \\
\mathsf{determine}(\texttt{LogicOp}(e_1, e_2)) &= 1 \\
\mathsf{determine}(\texttt{ReductionOp}(e)) &= 1 \\
\mathsf{determine}(\texttt{ShiftOp}(e_1, e_2)) &= \mathsf{determine}(e_1) \\
\mathsf{determine}(\texttt{AssignmentOp}(l, e)) &= \phi(l) \\
\mathsf{determine}(\texttt{ConditionalOp}(e_1, e_2, e_3) &= \max(\mathsf{determine}(e_2), \mathsf{determine}(e_3)) \\
\mathsf{determine}(\texttt{Replication}(n, e)) &= n \times \mathsf{determine}(e) \\
\mathsf{determine}(\texttt{Concatenation}(e_1, \ldots, e_k)) &= \sum_{i=1}^{k} \mathsf{determine}(e_i)
\end{aligned}
$$

**Figure 5.** Our determine function inspired from Table 11-21 in [16, §I.11.6.1]

*The Propagate Phase.* The propagate phase operates top-down, as described earlier (see examples in figures 2 and 3). Starting from the result of the determine phase, it flows bit-width information back into the expression tree to compute the final width of each sub-expression. While not explicitly defined in the LRM, this phase is implied by various requirements scattered throughout the specification.

Let $e$ be an expression. We model this phase as a function $\mathsf{propagate}_e : \mathrm{Path}(e) \to \mathbb{N}$ that maps each valid path in $e$ to the final width of the corresponding sub-expression. A path is represented as a list of natural numbers encoding navigation through the abstract syntax tree: starting from the root, each number indicates which child to visit next (0 denotes the first child, 1 the second, and so on). For example, the path $[1, 0]$ refers to the first child of the second child of the root expression. The empty path [] refers to the root expression itself. We use the notation $p \cdot k$ to denote extending path $p$ with index $k$ (the 'snoc' operation). We denote by $e|_p$ the sub-expression of $e$ reached by following the path $p$. Figure 6 shows all constraints mandated by the LRM for $\mathsf{propagate}_e$.

$$\mathsf{propagate}_e([]) = \mathsf{determine}(e) \tag{1}$$

For $e|_p = \texttt{ReductionOp}(e')$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{determine}(e') \tag{2}$$

For $e|_p = \texttt{LogicOp}(e_0, e_1)$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{determine}(e_0) \tag{3}$$
$$\mathsf{propagate}_e(p \cdot 1) = \mathsf{determine}(e_1) \tag{4}$$

For $e|_p = \texttt{Concatenation}(e_0, \ldots, e_k)$, with $i \in \{0, \ldots, k\}$:

$$\mathsf{propagate}_e(p \cdot i) = \mathsf{determine}(e_i) \tag{5}$$

For $e|_p = \texttt{Replication}(n, e')$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{determine}(e') \tag{6}$$

For $e|_p = \texttt{ComparisonOp}(e_0, e_1)$:

$$\mathsf{propagate}_e(p \cdot 0) = \max(\mathsf{determine}(e_0), \mathsf{determine}(e_1)) \tag{7}$$
$$\mathsf{propagate}_e(p \cdot 1) = \max(\mathsf{determine}(e_0), \mathsf{determine}(e_1)) \tag{8}$$

For $e|_p = \texttt{BinaryOp}(e_0, e_1)$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{propagate}_e(p) \tag{9}$$
$$\mathsf{propagate}_e(p \cdot 1) = \mathsf{propagate}_e(p) \tag{10}$$

For $e|_p = \texttt{UnaryOp}(e')$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{propagate}_e(p) \tag{11}$$

For $e|_p = \texttt{ShiftOp}(e_0, e_1)$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{propagate}_e(p) \tag{12}$$
$$\mathsf{propagate}_e(p \cdot 1) = \mathsf{determine}(e_1) \tag{13}$$

For $e|_p = \texttt{ConditionalOp}(e_0, e_1, e_2)$:

$$\mathsf{propagate}_e(p \cdot 0) = \mathsf{determine}(e_0) \tag{14}$$
$$\mathsf{propagate}_e(p \cdot 1) = \mathsf{propagate}_e(p) \tag{15}$$
$$\mathsf{propagate}_e(p \cdot 2) = \mathsf{propagate}_e(p) \tag{16}$$

For $e|_p = \texttt{AssignmentOp}(l, e')$:

$$\mathsf{propagate}_e(p \cdot 0) = \max(\mathsf{determine}(e'), \phi(l)) \tag{17}$$

**Figure 6.** LRM-derived constraints defining our $\mathsf{propagate}_e$ function.

Section §I.11.8.2 of the LRM states:

> Propagate the type and size of the expression (or *self-determined* subexpression) back down [...] the expression. [16, §I.11.8.2]

From this statement, we derive our first constraint, equation (1), which applies to the top-level expression. Furthermore, when a sub-expression at position $p$ within an expression $e$ is *self-determined*, the constraint given in equation (18) applies:

$$\mathsf{propagate}_e(p) = \mathsf{determine}\,(e|_p) \tag{18}$$

This principle serves as the basis for the constraints presented in equations (2) to (6), since Table 11–21 identifies these child expressions as self-determined. In particular, equation (2) is a specific instance of equation (18) for the child of a reduction operator. In this case, the reduction operator itself is located at position $p$, its child expression, $e'$ at position $p \cdot 0$, and we have $e|_{p\cdot 0} = e'$.

Comparison operations require special treatment. Table 11-21 specifies that 'Operands are sized to $\max(L(\mathtt{i}), L(\mathtt{j}))$', where `i` and `j` denote the operands and $L$ represents the determine function. This yields the constraints in equations (7) and (8).

We assume that operands not marked as *self-determined* in Table 11-21 are *context-determined*. For operations with such operands, the LRM states:

> In general, any *context-determined* operand of an operator shall be the same type and size as the result of the operator. [16, §I.11.8.2]

This constraint applies to binary and unary operations (equations (9) to (11)) and extends naturally to operators with mixed operand types, such as shifts and conditionals (equations (12) to (16)).

Assignment expressions require careful treatment. The LRM specifies:

> When the right-hand side evaluates to fewer bits than the left-hand side, the right-hand side value is padded to the size of the left-hand side.
> [...]
> If the width of the right-hand expression is larger than the width of the left-hand side in an assignment, the MSBs of the right-hand expression shall be discarded to match the size of the left-hand side. [16, §I.10.7]

These requirements imply that the maximum of the left-hand side bit-width and the right-hand side's self-determined width propagates downward. When truncation occurs, it happens at the top level ('the MSBs of the right-hand expression shall be discarded'), treating the expression as self-determined. These properties yield the constraint formalized in equation (17).

From the constraints of figure 6, we verified in Rocq that $\mathsf{propagate}_e$ is uniquely defined for every sub-expression of any expression.

## 3 Bidirectional Typing Framework

We now present our bidirectional typing [21] system, designed to capture the expression bit-width computation of SystemVerilog expressions. Bidirectional typing combines two modes of typing: type checking, which checks that a program $e$ in a context $\Gamma$ has type $\tau$ ($\Gamma \vdash e \Leftarrow \tau$), and type synthesis, which determines a type $\tau$ from the program $e$ in a context $\Gamma$ ($\Gamma \vdash e \Rightarrow \tau$). Using checking enables bidirectional typing to support features for which inference is undecidable; using synthesis enables bidirectional typing to avoid the large annotation burden of explicitly typed languages [10]. This technique has been used for dependent types [5], subtyping [21], polymorphism [19], and object-oriented languages including C# [1]. The computation of bit-widths in SystemVerilog presents a particularly suitable application domain for bidirectional typing due to the natural subtyping relation on bit-widths. We introduce two complementary typing judgments:

- The *synthesis* judgment, written $e \Rightarrow n \dashv f$, states that expression $e$ has a *self-determined width* $n$, where $f$ maps each sub-expression of $e$ to its *final width*.
- The *checking* judgment, written $e \Leftarrow n \dashv f$, states that expression $e$ *may be resized* to bit-width $n$, where $f$ maps each sub-expression of $e$ to its *final width*.

These judgments work in tandem: the synthesize judgment computes intrinsic properties of expressions (their self-determined widths), while the check judgment propagates width constraints down the AST, imposing bit-width on sub-expressions.

In both judgments, the function $f$ serves as a record of the typing decisions made for all sub-expressions, including the root one. Formally, $f : \mathrm{Path}(e) \to \mathbb{N}$ maps each valid path in expression $e$ to the final width of the corresponding sub-expression. This path-based representation allows us to precisely track the final width of every sub-expression within an expression tree. This capability is essential for proving the equivalence between our typing system and the current LRM specification.

$$\mathrm{Unary}(f,t) ::= \begin{cases} [] & \mapsto t \\ 0 \cdot p & \mapsto f(p) \end{cases} \qquad \mathrm{Binary}(t,f,g) ::= \begin{cases} [] & \mapsto t \\ 0 \cdot p & \mapsto f(p) \\ 1 \cdot p & \mapsto g(p) \end{cases}$$

$$\mathrm{Ternary}(t,f,g,h) ::= \begin{cases} [] & \mapsto t \\ 0 \cdot p & \mapsto f(p) \\ 1 \cdot p & \mapsto g(p) \\ 2 \cdot p & \mapsto h(p) \end{cases} \quad \mathrm{Nary}(t,f_1,\ldots,f_k) ::= \begin{cases} [] & \mapsto t \\ i \cdot p & \mapsto f_i(p) \end{cases}$$

**Figure 7.** Typing function combinators

To construct these functions during the typing process, we define the combinators shown in figure 7. These combinators construct typing functions from a bit-width $t$ (representing the final width of the current expression $e$) and one or more functions $\text{Path}(i) \to \mathbb{N}$ (representing the typing history of child expressions). The result is a complete typing function $\text{Path}(e) \to \mathbb{N}$ for the entire expression.

We present the typing rules by first explaining how expressions are resized in SystemVerilog. In the following section, we discuss how the self-determined width of an expression is computed.

### 3.1 Context-Dependent Resizing Rules

This section explains how expressions are resized when their context requires it. We focus on implicit resizing, not explicit resize casts. This operation preserves all information: expressions are never truncated, only extended when resized.

In SystemVerilog, resizing occurs at the deepest possible nodes in the AST, meaning resize operations propagate downward through the tree. We organize expressions into two categories based on their resizing behavior.

*Atomically Resizable Expressions.* They may be resized without affecting their internal operand widths. These include operands, comparisons, logical expressions, reductions, assignments, concatenations and replications. We denote the set of atomically resizable expressions as $\mathcal{R}$.

When an atomically resizable expression is resized to its final width, only the expression's result is extended, its operands remain unchanged. The corresponding typing rule is $\mathsf{Resize}\Leftarrow$, shown in figure 8.

To resize expression $e$ to the bit-width $t$, we verify that its self-determined width $s$ is no larger than $t$. This rule parallels bidirectional typing in subtyping systems [11], where a bit-vector of width $s$ acts as a subtype of a bit-vector of width $t$ when $s \leqslant t$.

The bit-width mapping $f$ is updated to reflect the new width, but since resizing atomically resizable expressions leaves their operands unchanged, only the empty path requires updating.

*Propagating Resize Operations.* Certain expressions propagate their target width to some or all of their operands. These operations correspond to expressions with *context-determined* arguments. The formal typing judgments for the propagating statements are presented in figure 8. We distinguish four cases:

- **Binary Operations** propagate the target width to both operands, as shown by the $\mathsf{BinOp}\Leftarrow$ rule.
- **Unary Operations** propagate the target width to their single operand, as shown by the $\mathsf{UnOp}\Leftarrow$ rule.
- **Shift Operations** propagate the target width only to the left operand, while the right operand is typed to its self-determined width, as shown by the $\mathsf{Shift}\Leftarrow$ rule.

– **Conditional Operations** propagate the target width to both branches, while the condition is typed to its self-determined width, as shown by the Cond⇐ rule.

These four cases encompass all non-atomically resizable expressions in SystemVerilog's expression typing system.

### 3.2 Computing *Self-Determined* Width

This section presents the rules for computing the self-determined width of expressions, which represents the natural bit-width an expression would have without any external contextual constraints. The complete set of synthesis rules is presented in figure 9.

The most critical aspect of self-determined width computation involves operations that require multiple typing judgments. Binary operations, comparisons, assignments, and conditionals each require two rules because one of their operands may be resized to match the other's self-determined width.

Consider binary operations, which must ensure both operands have the same width for evaluation. Since operands can only grow, the result width must be the maximum of both operand bit-width. When the left-hand side is larger, we apply the LBinOp⇒ rule, which synthesizes the self-determined width of the left operand $t$ and requires that the right operand may be resized to $t$. Conversely, when the right-hand side is larger, we apply the RBinOp⇒ rule, which synthesizes the self-determined width of the right operand and requires that the left operand may be resized accordingly.

The same pattern applies to comparisons and conditionals, though with different result widths. Comparisons always produce 1-bit results despite requiring operand width matching, while conditionals must ensure both branches have compatible widths.

Assignment operations demonstrate additional complexity. The RAssign⇒ rule applies when the right-hand side is strictly larger than the left-hand side of the assignment. In this case, and only in this case within SystemVerilog expression typing, the upper bits of the right-hand side are discarded to match the left-hand side's smaller width. The alternative rule, LAssign⇒, applies when the right-hand side may be resized to the left-hand side width. In both cases, the self-determined width of an assignment is determined by the left-hand side width.

Operations with asymmetric semantics, such as shifts and logical operations, have simpler synthesis rules. Shift operations inherit their width from the left operand only (Shift⇒), while logical operations always produce 1-bit results regardless of operand widths (Logic⇒).

Structural operations such as replication and concatenation have straightforward width computations: replication multiplies the operand size by the replication count (Repl⇒), while concatenation sums all operand widths (Concat⇒).

$$\frac{e \Rightarrow s \dashv f \qquad s \leqslant t \qquad e \in \mathcal{R}}{e \Leftarrow t \dashv f[[] \mapsto t]}\ \textsf{Resize}\Leftarrow$$

$$\frac{e \Leftarrow t \dashv f}{\oplus e \Leftarrow t \dashv \text{Unary}(t, f)}\ \textsf{UnOp}\Leftarrow \qquad \frac{e \Rightarrow t_e \dashv f_e \qquad a \Leftarrow t \dashv f_a \qquad b \Leftarrow t \dashv f_b}{e?a:b \Leftarrow t \dashv \text{Ternary}(t, f_e, f_a, f_b)}\ \textsf{Cond}\Leftarrow$$

$$\frac{a \Leftarrow t \dashv f_a \qquad b \Leftarrow t \dashv f_b}{a \oplus b \Leftarrow t \dashv \text{Binary}(t, f_a, f_b)}\ \textsf{BinOp}\Leftarrow \qquad \frac{a \Leftarrow t \dashv f_a \qquad b \Rightarrow t_b \dashv f_b}{a \oplus b \Leftarrow t \dashv \text{Binary}(t, f_a, f_b)}\ \textsf{Shift}\Leftarrow$$

**Figure 8.** *Resizing* typing rules

$$\frac{\Gamma(e) = s \qquad e \in \mathcal{O}}{e \Rightarrow s \dashv \{[] \mapsto s\}}\ \textsf{Operand}\Rightarrow$$

$$\frac{e \Rightarrow t \dashv f}{\oplus e \Rightarrow t \dashv \text{Unary}(t, f)}\ \textsf{UnOp}\Rightarrow \qquad \frac{e \Rightarrow t \dashv f}{\oplus e \Rightarrow 1 \dashv \text{Unary}(1, f)}\ \textsf{Red}\Rightarrow$$

$$\frac{a \Rightarrow t \dashv f_a \qquad b \Rightarrow t_b \dashv f_b}{a \oplus b \Rightarrow t \dashv \text{Binary}(t, f_a, f_b)}\ \textsf{Shift}\Rightarrow \qquad \frac{a \Rightarrow t_a \dashv f_a \qquad b \Rightarrow t_b \dashv f_b}{a \oplus b \Rightarrow 1 \dashv \text{Binary}(1, f_a, f_b)}\ \textsf{Logic}\Rightarrow$$

$$\frac{a \Rightarrow t \dashv f_a \qquad b \Leftarrow t \dashv f_b}{a \oplus b \Rightarrow t \dashv \text{Binary}(t, f_a, f_b)}\ \textsf{LBinOp}\Rightarrow \qquad \frac{a \Leftarrow t \dashv f_a \qquad b \Rightarrow t \dashv f_b}{a \oplus b \Rightarrow t \dashv \text{Binary}(t, f_a, f_b)}\ \textsf{RBinOp}\Rightarrow$$

$$\frac{a \Rightarrow t \dashv f_a \qquad b \Leftarrow t \dashv f_b}{a \oplus b \Rightarrow 1 \dashv \text{Binary}(1, f_a, f_b)}\ \textsf{LCmp}\Rightarrow \qquad \frac{a \Leftarrow t \dashv f_a \qquad b \Rightarrow t \dashv f_b}{a \oplus b \Rightarrow 1 \dashv \text{Binary}(1, f_a, f_b)}\ \textsf{RCmp}\Rightarrow$$

$$\frac{\phi(l) = t \qquad e \Leftarrow t \dashv f}{(l = e) \Rightarrow t \dashv \text{Unary}(t, f)}\ \textsf{LAssign}\Rightarrow \qquad \frac{\phi(l) = t \qquad e \Rightarrow t_e \dashv f \qquad t < t_e}{(l = e) \Rightarrow t \dashv \text{Unary}(t, f)}\ \textsf{RAssign}\Rightarrow$$

$$\frac{e \Rightarrow t_e \dashv f_e \qquad a \Rightarrow t \dashv f_a \qquad b \Leftarrow t \dashv f_b}{e?a:b \Rightarrow t \dashv \text{Ternary}(t, f_e, f_a, f_b)}\ \textsf{LCond}\Rightarrow$$

$$\frac{e \Rightarrow t_e \dashv f_e \qquad a \Leftarrow t \dashv f_a \qquad b \Rightarrow t \dashv f_b}{e?a:b \Rightarrow t \dashv \text{Ternary}(t, f_e, f_a, f_b)}\ \textsf{RCond}\Rightarrow$$

$$\frac{i \in \mathbb{N} \qquad e \Rightarrow t_e \dashv f \qquad t = i \times t_e}{\{i\ e\} \Rightarrow t \dashv \text{Unary}(t, f)}\ \textsf{Repl}\Rightarrow$$

$$\frac{e_1 \Rightarrow t_1 \dashv f_i \qquad \ldots \qquad e_k \Rightarrow t_k \dashv f_k \qquad t = t_1 + \cdots + t_k}{\{e_1, \ldots, e_k\} \Rightarrow t \dashv \text{Nary}(t, f_1, \ldots, f_k)}\ \textsf{Concat}\Rightarrow$$

**Figure 9.** *Self-determined width* typing rules

### 3.3 Soundness and Completeness Properties

Our typing system satisfies several fundamental properties that establish its soundness and completeness. All properties presented here have been mechanically verified in the Rocq theorem prover [23] with the functional extensionality axiom.

*Totality.* Every SystemVerilog expression can be typed in both synthesis and checking modes:

**Lemma 1 (Universal Typing).** *For every expression $e$:*

1. *There exist $t$ and $f$ such that $e \Rightarrow t \dashv f$*
2. *There exist $t$ and $f$ such that $e \Leftarrow t \dashv f$*

This property ensures our typing system is complete, it can assign bit-widths to any syntactically valid SystemVerilog expression.

*Determinism.* The typing system produces unique results for each expression:

**Lemma 2 (Typing Determinism).** *For any expression $e$:*

1. *If $e \Rightarrow t_1 \dashv f_1$ and $e \Rightarrow t_2 \dashv f_2$, then $t_1 = t_2$ and $f_1 = f_2$*
2. *If $e \Leftarrow t \dashv f_1$ and $e \Leftarrow t \dashv f_2$, then $f_1 = f_2$*

The synthesis mode uniquely determines both the self-determined width and the complete typing function. The checking mode, given a target width, uniquely determines how sub-expressions are sized.

*Synthesis-Checking Correspondence.* The two typing modes are related through a fundamental correspondence that characterizes when an expression may be resized:

**Lemma 3 (Synthesis-Checking Equivalence).** *If $e \Rightarrow t \dashv f$, then for any width $s$:*

$$t \leqslant s \iff \exists g.\, e \Leftarrow s \dashv g$$

This theorem states that an expression may be resized to $s$ if and only if $s$ is at least as large as its self-determined width. This captures the fundamental principle that expressions in SystemVerilog can only grow, never shrink.

*Minimality.* The self-determined width represents the minimal width at which an expression may be resized:

**Theorem 1 (Synthesis as Minimal Width).** *Let $e$ be an expression. If $s$ is the minimal width such that there exists $g$ where $e \Leftarrow s \dashv g$, then there exists $f$ such that $e \Rightarrow s \dashv f$.*

This property establishes that the self-determined width is not arbitrarily determined but represents the minimal bit-width that preserves information throughout the expression's evaluation. This ensures that expressions are dimensioned with sufficient width to maintain expected operational semantics while simultaneously minimizing the circuitry footprint of the design. This follows from lemmas 1 and 3.

These properties provide a solid theoretical foundation for our typing system, ensuring it captures the intended properties of the LRM specification.

## 4 Formal Verification of LRM Compliance

Establishing backward compatibility with the current LRM specification is essential for the adoption of our formal framework. To achieve this, we prove that our bidirectional typing system is equivalent to the algorithmic description scattered throughout the LRM. To demonstrate this correspondence, we prove that for any expression $e$, the typing function $f$ produced by our synthesis judgment $e \Rightarrow t \dashv f$ coincides with the $\mathsf{propagate}_e$ function derived from the LRM specification. The complete mechanized proofs can be interactively browsed online [9].

*Sub-expression Correspondence.* We establish the following fundamental lemmas:

**Lemma 4 (Sub-expression Synthesis).** *If $e \Rightarrow t \dashv f$ and $e|_p = e'$ for some path $p$, then there exist $t'$ and $f'$ such that either $e' \Rightarrow t' \dashv f'$ or $e' \Leftarrow t' \dashv f'$, and for all paths $k$: $f(p \cdot k) = f'(k)$.*

**Lemma 5 (Sub-expression Checking).** *If $e \Leftarrow t \dashv f$ and $e|_p = e'$ for some path $p$, then there exist $t'$ and $f'$ such that either $e' \Rightarrow t' \dashv f'$ or $e' \Leftarrow t' \dashv f'$, and for all paths $k$: $f(p \cdot k) = f'(k)$.*

These lemmas establish that typing functions compose properly: the typing function for a sub-expression can be extracted from the parent's typing function by shifting paths appropriately. This compositional property is essential for relating our typing system to the LRM's recursive structure.

**Lemma 6 (Synthesis Computes Determine).** *For every expression $e$, there exists a typing function $f$ such that $e \Rightarrow \mathsf{determine}(e) \dashv f$.*

This lemma ensures that our synthesis judgment correctly computes what the standard defines as the *self-determined width*, establishing the foundation for our equivalence proof.

*From Specification to Typing System.* We first prove that the function satisfying the LRM's propagate constraints correspond to the valid typing derivation:

**Lemma 7 (Specification Implies Typing).** *For any expression $e$ and function $f : \mathrm{Path}(e) \to \mathbb{N}$, if $f$ satisfies all LRM propagate constraints, then $e \Rightarrow \mathsf{determine}(e) \dashv f$.*

The proof proceeds by path induction. We leverage lemma 6 to obtain an initial typing derivation, then use lemma 4 and lemma 5 to show that the typing function matches $\mathsf{propagate}_e$ at each path. This correspondence is possible because both the typing system and the LRM specification propagate widths using identical structural rules for *context-determined* operands.

*From Typing System to Specification.* The converse direction establishes that our typing system generates the function that satisfy all LRM constraints:

**Lemma 8 (Typing Implies Specification).** *For any expression $e$, typing function $f$, and width $t$, if $e \Rightarrow t \dashv f$ holds, then $f$ satisfies all LRM propagate constraints.*

This proof proceeds by case analysis, verifying that each propagate constraint from the LRM is preserved.

*Main Equivalence Result.* Combining lemma 7 and lemma 8, we obtain our main equivalence result:

**Theorem 2 (Typing System–LRM Equivalence).** *For any expression $e$ and function $f : \mathrm{Path}(e) \to \mathbb{N}$, the two following statements are equivalent:*

1. *$f$ satisfies all LRM propagate constraints for expression $e$.*
2. *$e \Rightarrow \mathsf{determine}(e) \dashv f$.*

This equivalence, mechanically verified in Rocq, establishes that our bidirectional typing system provides a precise mathematical characterization of the LRM's bit-width determination algorithm. The proof demonstrates that our formal framework preserves existing SystemVerilog semantics while offering a cleaner mathematical foundation suitable for formal verification and tool development.



## 5 Implementation and Complexity

Building upon our bidirectional typing system, we derive a concrete algorithm for computing expression bit-widths that closely follows the LRM's two-phase approach. Algorithm 1 computes the *self-determined width* bottom-up and algorithm 2 propagates width information top-down to determine the *final width* of each sub-expression. Typing a top-level expression $e$ is done with the expression $\textsc{propagate}(e, \textsc{determine}(e))$.

Our implementation has been mechanically verified in Rocq and can be extracted from our formalization to produce executable code. The complete source code for our formalization and extracted algorithms is publicly available [8].

**Algorithm 1:** Determine

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

```
1  switch expr do
2      when expr is an operand do
3          return Γ(expr)
4      when expr is lhs ⊕ rhs do                    // ⊕ can be +, -, *, /, ...
5          lhs_w ← DETERMINE(lhs)
6          rhs_w ← DETERMINE(rhs)
7          return max(lhs_w, rhs_w)
8      when expr is ⊕arg do                         // ⊕ can be +, -, ~, ++, --
9          arg_w ← DETERMINE(arg)
10         return arg_w
11     when expr is lhs ⊕ rhs do                    // ⊕ can be ==, !=, >, >=, ...
12         return 1
13     when expr is lhs ⊕ rhs do                    // ⊕ can be &&, ||, ->, <->
14         return 1
15     when expr is ⊕arg do                         // ⊕ can be !, &, |, ^, ...
16         return 1
17     when expr is lhs ⊕ rhs do                    // ⊕ can be >>, <<, **, ...
18         lhs_w ← DETERMINE(lhs)
19         return lhs_w
20     when expr is lval = rhs do
21         lval_w ← φ(lval)
22         return lval_w
23     when expr is cond ? lhs : rhs do
24         lhs_w ← DETERMINE(lhs)
25         rhs_w ← DETERMINE(rhs)
26         return max(lhs_w, rhs_w)
27     when expr is {expr_1, ... , expr_N} do
28         for i ∈ {1, ..., N} do
29             width_i ← DETERMINE(expr_i)
30         return Σ_{i=0}^{N} width_i
31     when expr is {n arg} do
32         arg_w ← DETERMINE(arg)
33         return n × 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 ⊕ rhs do                    // ⊕ can be +, -, *, /, ...
5          PROPAGATE(lhs, targetWidth)
6          PROPAGATE(rhs, targetWidth)
7          Annotate expr with targetWidth
8      when expr is ⊕arg do                         // ⊕ can be +, -, ~, ++, --
9          PROPAGATE(arg, targetWidth)
10         Annotate expr with targetWidth
11     when expr is lhs ⊕ rhs do                    // ⊕ can be ==, !=, >, >=, ...
12         arg_w ← max (DETERMINE(lhs), DETERMINE(rhs))
13         PROPAGATE(lhs, arg_w)
14         PROPAGATE(rhs, arg_w)
15         Annotate expr with targetWidth
16     when expr is lhs ⊕ rhs do                    // ⊕ can be &&, ||, ->, <->
17         PROPAGATE(lhs, DETERMINE(lhs))
18         PROPAGATE(rhs, DETERMINE(rhs))
19         Annotate expr with targetWidth
20     when expr is ⊕arg do                         // ⊕ can be !, &, |, ^, ...
21         PROPAGATE(arg, DETERMINE(arg))
22         Annotate expr with targetWidth
23     when expr is lhs ⊕ rhs do                    // ⊕ 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 (φ(lval), DETERMINE(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 {expr_1, ... , expr_N} do
36         for i ∈ {1, ..., N} do
37             PROPAGATE(expr_i, DETERMINE(expr_i))
38         Annotate expr with targetWidth
39     when expr is {n arg} do
40         PROPAGATE(arg, DETERMINE(arg))
41         Annotate expr with targetWidth
```

### 5.1 Algorithm Verification

The key correctness result establishes that PROPAGATE correctly encompasses the checking judgment:

**Lemma 9 (Propagate/Check Correspondence).** *For any expression $e$ and width $s$ where $\mathsf{determine}(e) \leqslant s$, the call $\textsc{propagate}(e, s)$ produces annotations such that the corresponding typing function $f$ satisfies $e \Leftarrow s \dashv f$.*

The typing function $f$ is simply the lookup of annotations added during the PROPAGATE pass: for each path $p$, the value $f(p)$ is the annotation at that path. From this invariant, it follows that the overall algorithm is correct:

**Theorem 3 (Algorithm Correctness).** *For any expression $e$, the typing function $f$ resulting from the call to $\textsc{propagate}(e, \textsc{determine}(e))$ is such that $e \Rightarrow \mathsf{determine}(e) \dashv f$.*

This establishes that our two-phase algorithm correctly implements the synthesis judgment at the root expression, with all sub-expressions properly typed according to our formal system.

### 5.2 Complexity Analysis

The algorithm achieves linear time complexity in the size of the expression tree when DETERMINE calls are memoized. Without memoization, repeated calls to DETERMINE during PROPAGATE lead to quadratic behavior, for instance, on deeply nested concatenations where each level requires recomputing widths of all inner expressions. With memoization, each expression node is visited exactly twice: once during the bottom-up DETERMINE phase and once during the top-down PROPAGATE phase.

## 6 Standardization Proposal

Drawing from our formal framework, we developed a proposal [7] to clarify bit-vector expression bit-width in the LRM. The proposal addresses specification ambiguities while preserving backward compatibility.

Our contribution includes three components: rigorous typing rules accessible to engineers unfamiliar with formal methods, extensive examples demonstrating how they are used, and the verified algorithm as a reference implementation. This machine-checked algorithm provides tool developers with a reliable foundation for SystemVerilog expression typing.

## 7 Conclusion

We have presented a formal framework for SystemVerilog expression bit-width determination based on bidirectional typing. Our approach provides a precise

mathematical foundation for this essential aspect of the language while maintaining full backward compatibility with the IEEE 1800-2023 standard.

We demonstrated that the informal prose specifications scattered throughout the 1,354-page LRM can be captured by a clean mathematical framework. The bidirectional typing system naturally expresses the context-dependent nature of SystemVerilog expression sizing through the interplay of synthesis and checking judgments. The mechanized proofs in Rocq establish both the internal consistency of our typing system and its equivalence to the LRM specification.

Looking forward, this work lays the foundation for a fully verified SystemVerilog type checker. Future work includes extending the framework to handle signedness, all SystemVerilog types, and the full complexity of SystemVerilog's expression type system. This formal foundation could be used to develop verified tooling for the SystemVerilog ecosystem.

By bridging the gap between informal language specifications and formal methods, we hope to enable more reliable hardware design tools and ultimately more trustworthy digital systems.

## References


1. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. SIGPLAN Not. **42**(10), 479–498 (Oct 2007). https://doi.org/10.1145/1297105.1297063
2. Chen, Q., Zhang, N., Wang, J., Tan, T., Xu, C., Ma, X., Li, Y.: The essence of verilog: A tractable and tested operational semantics for verilog. Proc. ACM Program. Lang. **7**(OOPSLA2) (Oct 2023). https://doi.org/10.1145/3622805
3. ChipsAlliance: sv-tests: Test suite designed to check compliance with the SystemVerilog standard. GitHub repository, https://github.com/chipsalliance/sv-tests
4. Choi, J., Kim, J., Kang, J.: Revamping verilog semantics for foundational verification (2025), draft paper to appear at OOPSLA 2025.
5. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming **26**(1), 167–177 (1996). https://doi.org/10.1016/0167-6423(95)00021-6
6. Corradi, Q., Wickerson, J., Constantinides, G.A.: Automated feature testing of verilog parsers using fuzzing (registered report). In: Proceedings of the 3rd ACM International Fuzzing Workshop. p. 70–79. FUZZING 2024, Association for Computing Machinery, New York, NY, USA (Sep 2024). https://doi.org/10.1145/3678722.3685536
7. Desfrene, G., Corradi, Q., Pardalos, M., Wickerson, J.: Proposal for Improving Expression Size Determination in IEEE 1800 SystemVerilog (2025), https://raw.githubusercontent.com/desfreng/Verilog-Typing/main/Proposal/Proposal.pdf
8. Desfrene, G., Corradi, Q., Pardalos, M., Wickerson, J.: SystemVerilog Bidirectional Typing Implementation (2025), https://github.com/desfreng/Verilog-Typing
9. Desfrene, G., Corradi, Q., Pardalos, M., Wickerson, J.: SystemVerilog Expression Typing: Interactive Rocq Proofs (2025), https://gabriel.desfrene.fr/verilog/proof/
10. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5) (May 2021). https://doi.org/10.1145/3450952

11. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. SIGPLAN Not. **48**(9), 429–442 (Sep 2013). https://doi.org/10.1145/2544174.2500582
12. Flake, P., Moorby, P., Golson, S., Salz, A., Davidmann, S.: Verilog HDL and its ancestors and descendants. Proc. ACM Program. Lang. **4**(HOPL) (Jun 2020). https://doi.org/10.1145/3386337
13. Foster, H.D.: The 2022 wilson research group functional verification study (2022), https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study
14. Gillenwater, J., Malecha, G., Salama, C., Zhu, A.Y., Taha, W., Grundy, J., O'Leary, J.: Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee verilog synthesizability. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. p. 41–50. PEPM '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1328408.1328416
15. Herklotz, Y., Wickerson, J.: Finding and understanding bugs in fpga synthesis tools. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 277–287. FPGA '20, Association for Computing Machinery, New York, NY, USA (Feb 2020). https://doi.org/10.1145/3373087.3375310
16. IEEE: IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017) (Feb 2024). https://doi.org/10.1109/IEEESTD.2024.10458102
17. Lööw, A.: Lutsig: a verified Verilog compiler for verified circuit development. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 46–60. CPP 2021, Association for Computing Machinery, New York, NY, USA (Jan 2021). https://doi.org/10.1145/3437992.3439916
18. Lööw, A.: The simulation semantics of synthesisable verilog. Proc. ACM Program. Lang. **9**(OOPSLA1) (Apr 2025). https://doi.org/10.1145/3720484
19. Odersky, M., Zenger, C., Zenger, M.: Colored local type inference. SIGPLAN Not. **36**(3), 41–53 (Jan 2001). https://doi.org/10.1145/373243.360207
20. Pardalos, M., Pozzi, L., Wickerson, J.: Towards mechanized verification of Verilog equivalence checking. In: Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) (Mar 2025)
21. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. **22**(1), 1–44 (Jan 2000). https://doi.org/10.1145/345099.345100
22. Rich, D.: What's next for SystemVerilog in the upcoming IEEE 1800 standard (2023), https://resources.sw.siemens.com/en-US/white-paper-whats-next-for-system-verilog-in-the-upcoming-ieee-1800-standard/, white Paper
23. The Rocq Development Team: The Rocq Prover (2025), https://rocq-prover.org/, version 9.0.0
24. Vieira, J.V.A., Gomes, L.d.M., Sumitani, R., Maciel, R., Mafra, A., Crepalde, M., Pereira, F.M.Q.: Bottom-up generation of verilog designs for testing eda tools (Apr 2025), https://arxiv.org/abs/2504.06295