A mechanised, bidirectional type system for bit-width determination 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, which can lead to inconsistent implementations and portability challenges. Formal methods can provide precise semantics to address these issues.

We focus on SystemVerilog's mechanism for determining the bit-width of each expression that appears in a design. This is surprisingly subtle because an expression's bit-width can depend on both its children *and* its parents. First, we develop a Rocq formalization of the existing IEEE standard for SystemVerilog. We then construct a bidirectional type system that captures the context-dependent nature of SystemVerilog expressions and prove it equivalent to our formalization of the standard using Rocq. We provide a reference implementation that determines bit-widths in linear time and prove its correspondence to our system, also in Rocq. Based on these results, we propose revisions to the text of the standard that reduce redundancy and improve precision.

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 [9], enabling the design, verification, and synthesis of everything from simple logic gates to complex system-on-chip designs. But important aspects of the language remain poorly understood and inconsistently implemented [3], thanks to its large and cumbersome prose specification, the IEEE 1800-2023 SystemVerilog Language Reference Manual (LRM) [11].

For instance: because SystemVerilog designs map to physical hardware, the width in bits of every operation in every expression must be determined precisely. Too narrow, and arithmetic precision is lost; too wide, and the circuit will consume area unnecessarily. Some widths are provided by the designer via constants and

variable declarations; others are determined during compilation following rules set out in the standard.

However, these rules are complex and widely misunderstood. Consider the following example, in which `a` is declared as a 4-bit variable, `3'b110` is the 3-bit binary constant `110`, and `>>` is a right-shift operation:

```
logic [3:0] a;
assign a = (3'b110 + 3'b110 + 3'b110 + 3'b110) >> 2;
```

What value is assigned to `a`?

- One option is that the additions are performed at whatever width is necessary for no bits to be lost to overflow. In this case, the intermediate sum would be `5'b11000`, so after the right-shift, `a` would become `4'b0110`.
- A second option is to perform the additions at the width of their operands, which is 3 bits. In this case, the intermediate sum would be `3'b000`, so after the right-shift, `a` would become `4'b0000`.

In fact, SystemVerilog opts for a *third* option: the addition is performed at the width determined by the assignment's left-hand side (`a`), which is 4 bits. So, the intermediate sum is `4'b1000`, and hence after the right shift, `a` becomes `4'b0010`.

Further complexities arise when other operations are involved; for instance, although we see here widths propagating down through addition and shift operations, there are other operations, such as concatenation, that they do *not* propagate through.

While there are no documented cases of rockets having fallen to earth as a direct result of SystemVerilog's bit-width rules being misunderstood, there is plenty of anecdotal evidence of confusion surrounding them. For instance, an established online Verilog tutorial, after warning its readers that bit-width rules are 'subtle', wrongly states that operations have their widths determined by their operands (i.e., our 'second option' mentioned above) [19]. There have also been cases where companies have been so concerned about bugs relating to bit-widths that they have required their designers to make even benign zero-extensions explicit [15]. And there are known quirks in how commercial tools handle bit-widths: for instance, SystemVerilog forbids unsized constants appearing in concatenations, and Cadence tools duly reject expressions like `{1'b0, 16}`—yet, according to the developer of the Icarus Verilog simulator [21], they happily accept `{1'b0, 15+1}`!

In short, bit-width determination in SystemVerilog is complicated; partly because of the wide variety of rules for the different operations and partly because of the unusual context-dependent manner in which they are applied. Because of this, and because of the importance of correct hardware compilation, our position is that bit-width determination in SystemVerilog is an ideal candidate for formalisation.

Prior work. SystemVerilog has been the focus of several recent formalisation efforts, but none have addressed bit-width determination with sufficient rigour. Featherweight Verilog introduces a minimal core calculus for the synthesizable subset of Verilog [10], but relies on generic types and omits bit-width determination entirely. Choi et al.'s denotational semantics assumes bit-widths are fixed *a priori* [4], thereby sidestepping bit-width determination, while Lööw's analysis of inconsistencies in the SystemVerilog LRM focuses on simulation semantics rather than formalizing bit-width determination rules [13]. Other efforts aim to define executable or reference semantics for large fragments of SystemVerilog, but do not provide an efficient, mechanically verified procedure for computing bit-width information. Chen et al. propose a tractable operational semantics that computes bit-widths algorithmically [2], but their approach lacks formal judgments suitable for mechanized correctness proofs. Meredith et al.'s rewriting-based executable semantics covers a substantial portion of SystemVerilog [14], but does not support sized integers and fails to yield a clear or efficient decision procedure for bit-width determination.

Our work. This paper addresses this gap by developing the first verified system for determining SystemVerilog bit-widths. As mentioned above, what complicates matters is that in SystemVerilog, the bit-width of an 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, in which bit-width information first flows from the operands up to the root of the expression (the 'determine' phase), and then flows back down, fixing the final size of each sub-expression (the 'propagate' phase). Our approach resolves this interdependency in a principled way by setting up a bidirectional typing system [18,8], in which bidirectional typing's *synthesis* mode maps to the 'determine' phase, its *checking* mode maps to the 'propagate' phase, and a bit-vector of width s is defined to be a subtype of a bit-vector of width t whenever $s \leq t$. This typing system is proven equivalent to a formalisation of the LRM, but it is easier to use in a context of a proof assistant because it is fully compositional. It is also designed to be easily integrated (in future work) with existing formally verified Verilog-consuming tools, such as the Vera verified equivalence checker [17] and the Lustig verified synthesis tool [12], potentially enhancing the formal guarantees they provide.

Contributions. We claim the following contributions:

1. We formalize an important subset of the IEEE 1800-2023 specification for bit-width determination in the Rocq theorem prover[20], providing a precise mathematical interpretation of the standard's prose descriptions (section 2).
2. We develop a bidirectional type system that captures the context-dependent nature of SystemVerilog expression bit-widths, introducing formal rules and proving key properties (section 3).
3. We provide machine-checked proofs that our rule system is equivalent to our formalization of the LRM (section 4).

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 rule system (section 5).

Auxiliary material. Our online appendix [6] contains all implementations and Rocq proofs. It also presents our draft proposal to improve the treatment of bit-width determination in the IEEE 1800-2023 standard. This proposal is grounded in the formal framework developed in this work and aims to provide a comprehensive and unambiguous definition of bit-width determination in SystemVerilog.

2 Formalizing the LRM

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 approach to formalising it in Rocq.

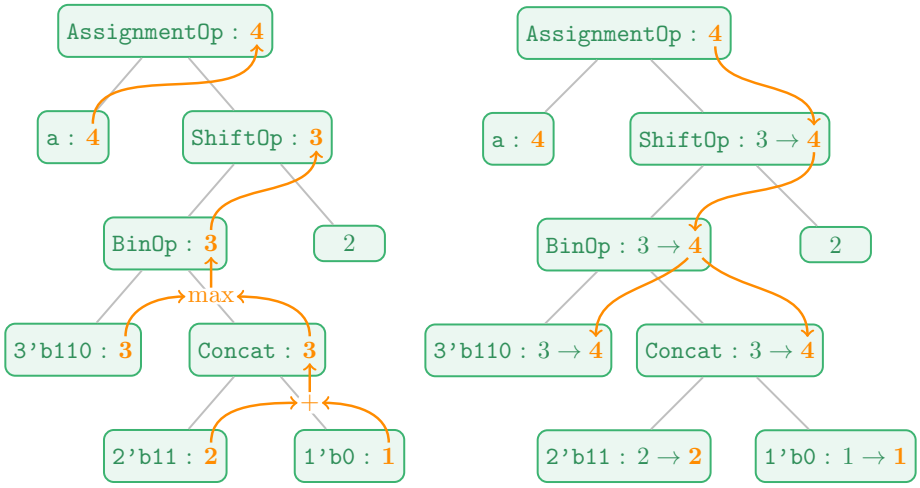
2.1 How Sizing Works in SystemVerilog

The *self-determined width* of an expression is the bit-width determined solely by the expression itself, independent of its context. This constitutes an intrinsic property of the expression, corresponding to what the LRM refers to in its latest version[11, §I.11.6] as a *self-determined* expression. Practically, this property ensures that the widths of such expressions 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-widths are explicitly known, then propagates upward through the abstract syntax tree according to operator-specific rules. Figure 1a illustrates this phase using the following example code (which is similar to the example from section 1 but involves fewer additions and includes a concatenation operation).

```
logic [3:0] a;
assign a = (3'b110 + {2'b11, 1'b0}) >> 2;
```

Each node of the AST is annotated with the kind of operation being performed (`BinOp` for +, and so on), and also its self-determined width. The bold orange arrows show how the self-determined widths are propagated *up* the AST: the width of the concatenation is the sum of the widths of its operands; the width of the binary operation is the maximum of the widths of its operands; the shift's width is determined by its left-hand operand (the value being shifted); and the width of the result of the assignment is determined by the width of its left-hand



(a) The Determine phase (b) The Propagate phase

Figure 1: An example of the two phases of bitwidth calculation

side (`a` in this case). Thus, the Determine phase computes the self-determined width of the top-level expression and all sub-expressions.

The self-determined width of the top-level expression also becomes its final width. Then the final widths of all the sub-expressions are calculated during a subsequent Propagate phase. This is illustrated in figure 1b. Here, nodes of the AST are annotated with two widths, $w \to v$: the self-determined width w and the final width v. The bold orange arrows show how the widths are propagated *down* the AST: the final width of the shift is determined by the width of its parent; as is the width of the binary operation and both of its operands. However, the width information does *not* flow down to the operands of the `Concat` operation: these belong to a separate 'context', so their final widths remain as their self-determined widths.

In summary, the complete sizing mechanism operates through a two-phase process. The first phase, Determine, synthesizes self-determined bit-widths throughout the expression hierarchy, plus the final width of the top-level expression. The second phase, Propagate, computes the final width for all sub-expressions by propagating bit-width information downward through the AST.

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 behaviours relevant to bit-width determination.

We begin with operands, which are the atomic parts of SystemVerilog expressions: literals, variables, structure members, union members, function calls, and

$e ::= o$		operands
|	`BinaryOp(`e_1`, `e_2`)`	binary and bitwise 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
|	`Concat(`e_1`, ..., `e_k`)`	concatenation
|	`Replication(`n`, `e`)`	replication

Figure 2: SystemVerilog expressions, categorized by bit-width determination behaviour

other primitive constructs such as bit selects and bit-slicing [11, §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 widths are defined by the function's return type. For bit-width generic functions such as type casts, the output width can be determined from the function signature and the self-determined widths from its arguments (which themselves require bit-width analysis). This normalization guarantees that all operands possess well-defined self-determined widths while simplifying subsequent operations.

We categorize SystemVerilog operations by their sizing behaviors, yielding the expression grammar shown in figure 2. 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.

A few Verilog-specific features are worth explaining. 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 bitvector 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`).

We remark on a few simplications in our grammar. First, compound assignment operators (`+=`, etc.) are treated as syntactic sugar (e.g. `a += b` is interpreted as `a = a + b`) as specified in the LRM[11, §I.11.4.1]. Second, the LRM constrains the `Replication` operator to operate solely on `Concat` expressions, but we adopt a more permissive approach by allowing replication of *any* expression, because it simplifies our inference rules. Third, 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 the LRM[11, §I.11.4.13]. Fourth, we exclude streaming operators from our formalization as they behave identically to concatenation with respect to bit-width determination. Finally, 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.

2.3 Formalizing the LRM

The LRM does not present a unified 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. [11, §I.11.8.2]

The LRM defines a *context-determined* expression as one whose bit length

> [...] is determined by the bit length of the expression and by the fact that it is part of another expression. [11, §I.11.6.1]

The Determine Phase. The first phase is well-defined by Table 11–21 in the LRM[11, §I.11.6.1]. This phase computes the self-determined width for each expression in a bottom-up fashion, as seen in figure 1a. We formalize this as a recursive function $\mathsf{det} : \text{Expr} \to \mathbb{N}$, defined in figure 3.

This formalization relies on two auxiliary functions, Γ and ϕ. The function Γ 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 ϕ determines the bit-width of left-hand sides in assignments. This function exists because assignment targets have a simpler syntactic structure than general expressions [11, §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.

The Propagate Phase. The Propagate phase operates top-down, as described earlier and illustrated in figure 1b. 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 (in §I.10.7, and from §I.11.6 to §I.11.8).

Let e be an expression. We model this phase as a function $\mathsf{prop}_e : \text{Path}(e) \to \mathbb{N}$ 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

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

Figure 3: Our det function, inspired by Table 11–21 in the LRM[11, §I.11.6.1]

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. We denote by $e|_p$ the sub-expression of e reached by following the path p. Figure 4 shows all constraints mandated by the LRM for prop_e.

Section §I.11.8.2 of the LRM states that this phase should:

> Propagate the type and size of the expression (or *self-determined* subexpression) back down [...] the expression. [11, §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 following constraint applies:

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

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 $\max\,(L(\texttt{i}), L(\texttt{j}))$', where i and j denote the operands and L represents the det 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 that:

> In general, any *context-determined* operand of an operator shall be the same type and size as the result of the operator. [11, §I.11.8.2]

$$\mathsf{prop}_e([]) = \mathsf{det}(e) \tag{1}$$

$$\text{For } e|_p = \texttt{ReductionOp}(e') : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{det}(e') \tag{2}$$

$$\text{For } e|_p = \texttt{LogicOp}(e_0, e_1) : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{det}(e_0) \tag{3}$$

$$\mathsf{prop}_e(p \cdot 1) = \mathsf{det}(e_1) \tag{4}$$

$$\text{For } e|_p = \texttt{Concat}(e_0, \ldots, e_k), 0 \leq i \leq k : \quad \mathsf{prop}_e(p \cdot i) = \mathsf{det}(e_i) \tag{5}$$

$$\text{For } e|_p = \texttt{Replication}(n, e') : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{det}(e') \tag{6}$$

$$\text{For } e|_p = \texttt{ComparisonOp}(e_0, e_1) : \quad \mathsf{prop}_e(p \cdot 0) = \max(\mathsf{det}(e_0), \mathsf{det}(e_1)) \tag{7}$$

$$\mathsf{prop}_e(p \cdot 1) = \max(\mathsf{det}(e_0), \mathsf{det}(e_1)) \tag{8}$$

$$\text{For } e|_p = \texttt{BinaryOp}(e_0, e_1) : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{prop}_e(p) \tag{9}$$

$$\mathsf{prop}_e(p \cdot 1) = \mathsf{prop}_e(p) \tag{10}$$

$$\text{For } e|_p = \texttt{UnaryOp}(e') : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{prop}_e(p) \tag{11}$$

$$\text{For } e|_p = \texttt{ShiftOp}(e_0, e_1) : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{prop}_e(p) \tag{12}$$

$$\mathsf{prop}_e(p \cdot 1) = \mathsf{det}(e_1) \tag{13}$$

$$\text{For } e|_p = \texttt{ConditionalOp}(e_0, e_1, e_2) : \quad \mathsf{prop}_e(p \cdot 0) = \mathsf{det}(e_0) \tag{14}$$

$$\mathsf{prop}_e(p \cdot 1) = \mathsf{prop}_e(p) \tag{15}$$

$$\mathsf{prop}_e(p \cdot 2) = \mathsf{prop}_e(p) \tag{16}$$

$$\text{For } e|_p = \texttt{AssignmentOp}(l, e') : \quad \mathsf{prop}_e(p \cdot 0) = \max(\mathsf{det}(e'), \phi(l)) \tag{17}$$

Figure 4: LRM-derived constraints defining our prop_e function.

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 that:

> 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. [11, §I.10.7]

These requirements imply that the maximum of the left-hand side's 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 4, we verified in Rocq that prop_e is uniquely defined for every sub-expression of any expression.

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

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

Figure 5: Sizing function combinators

3 Bidirectional Typing Framework

We now present our bidirectional type system [18,8], which is designed to capture the bit-width computation of SystemVerilog expressions. Bidirectional typing combines two modes: type *checking*, which checks that a program e in a context Γ has type τ (written $\Gamma \vdash e \Leftarrow \tau$), and type *synthesis*, which determines a type τ from the program e in a context Γ (written $\Gamma \vdash e \Rightarrow \tau$). Using checking enables bidirectional typing to support features for which determination is undecidable; using synthesis enables bidirectional typing to avoid the large annotation burden of explicitly typed languages[7]. This technique has been used for dependent types[5], subtyping[18], polymorphism[16], 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 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*.

In both judgments, the sizing function f serves as a record of the decisions made for all sub-expressions, including the root one. Formally, $f : \text{Path}(e) \to \mathbb{N}$ 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 type system and the current LRM specification.

To construct these functions during the inference process, we define the combinators shown in figure 5. These combinators construct sizing 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 sizing history of child expressions). The result is a complete sizing function $\text{Path}(e) \to \mathbb{N}$ for the entire expression.

We present the 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 rule is $\mathsf{Resize}\Leftarrow$, shown in figure 6. 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[8], 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 judgments for the propagating statements are presented in figure 6. 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 fixed to its self-determined width, as shown by the $\mathsf{Shift}\Leftarrow$ rule; and
- conditional operations propagate the target width to both branches, while the condition is fixed to its self-determined width, as shown by the $\mathsf{Cond}\Leftarrow$ rule.

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

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

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

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

Figure 6: *Resizing* rules

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

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

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

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

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

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

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

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

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

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

Figure 7: *Self-determined width* rules

3.2 Computing *Self-Determined* Width

This subsection 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 7.

The most critical aspect of self-determined width computation involves operations that require multiple 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 $\mathsf{LBinOp}\Rightarrow$ 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 $\mathsf{RBinOp}\Rightarrow$ 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 $\mathsf{RAssign}\Rightarrow$ rule applies when the right-hand side is strictly larger than the left-hand side of the assignment. In this case, the upper bits of the right-hand side are discarded to match the left-hand side's smaller width. The alternative rule, $\mathsf{LAssign}\Rightarrow$, 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 ($\mathsf{Shift}\Rightarrow$), while logical operations always produce 1-bit results regardless of operand widths ($\mathsf{Logic}\Rightarrow$). Structural operations such as replication and concatenation have straightforward width computations: replication multiplies the operand size by the replication count ($\mathsf{Repl}\Rightarrow$), while concatenation sums all operand widths ($\mathsf{Concat}\Rightarrow$).

3.3 Soundness and Completeness Properties

Our system satisfies several fundamental properties that establish its soundness and completeness. All properties presented here have been mechanically verified in the Rocq theorem prover[20] with the functional extensionality axiom.

Totality. Every SystemVerilog expression can be used in both synthesis and checking modes. This property ensures our type system is complete: it can assign bit-widths to any syntactically valid SystemVerilog expression.

Lemma 1. *For every expression e, (1) there exist t and f such that $e \Rightarrow t \dashv f$, and (2) there exist t and f such that $e \Leftarrow t \dashv f$.*

Determinism. The type system produces unique results for each expression. That is, the synthesis mode uniquely determines both the self-determined width and the complete sizing function, and the checking mode, given a target width, uniquely determines how sub-expressions are sized.

Lemma 2. *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 $f_1 = f_2$, and (2) if $e \Leftarrow t \dashv f_1$ and $e \Leftarrow t \dashv f_2$, then $f_1 = f_2$.*

Synthesis-Checking Correspondence. 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.

Lemma 3. *If $e \Rightarrow t \dashv f$, then for any width s, we have $t \leq s$ if and only if $\exists g.\, e \Leftarrow s \dashv g$.*

Minimality. The following theorem, which follows from lemmas 1 and 3, states that the self-determined width represents the minimal width at which an expression may be resized without losing information.

Theorem 1. *For any expression e: if s is the minimal width such that there exists g where $e \Leftarrow s \dashv g$, then there exists f such that $e \Rightarrow s \dashv f$.*

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 type system is equivalent to the algorithmic description scattered throughout the LRM. To demonstrate this correspondence, we prove that for any expression e, the function f produced by our synthesis judgment $e \Rightarrow t \dashv f$ coincides with the prop_e function derived from the LRM specification. The complete mechanized proofs can be interactively browsed online[6].

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

Lemma 4. *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. *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: $f(p \cdot k) = f'(k)$.*

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

Lemma 6. *For every expression e, there exists a sizing function f such that $e \Rightarrow \mathsf{det}(e) \dashv f$.*

From Specification to a Rule System. We first prove that the function satisfying the LRM's propagation constraints correspond to a valid derivation:

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

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

Lemma 8. *For any expression e, sizing function f, and width t, if $e \Rightarrow t \dashv f$ holds, then f satisfies all LRM propagate constraints.*

Main Equivalence Result. Combining lemma 7 and lemma 8, we obtain our main equivalence result: that our bidirectional type system provides a precise mathematical characterization of the LRM's bit-width determination algorithm.

Theorem 2. *For any expression e and function $f : \mathrm{Path}(e) \to \mathbb{N}$, the following two statements are equivalent: (1) f satisfies all LRM propagation constraints for expression e, and (2) $e \Rightarrow \mathsf{det}(e) \dashv f$.*

5 Implementation

Building upon our bidirectional type 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. Computing the width of 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 [6].

In a sense, we have almost come full circle: DETERMINE and PROPAGATE closely resemble det and prop from our original formalisation of the LRM (section 2). But the type system of section 3 remains a valuable intermediate step for proving the LRM and this implementation (as well as other possible future implementations) equivalent. This is partly because the type system is compositional—it does not need to carry around the top-level expression e like prop does—and partly because typing rules are very natural to work with in Rocq.

Correctness The following theorem establishes that our two-phase algorithm correctly implements the synthesis judgement at the root expression with all sub-expressions properly typed according to our formal system.

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

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
```

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 Conclusion and Future Directions

We have presented a formal framework for SystemVerilog expression bit-width determination based on bidirectional typing. We have demonstrated that the informal prose specifications scattered throughout 25 pages of the 1,354-page LRM can be captured by a clean mathematical framework, while maintaining full backward compatibility with the IEEE 1800-2023 standard. Our bidirectional type system naturally expresses the context-dependent nature of SystemVerilog expression sizing through the interplay of synthesis and checking judgments. Our mechanized proofs in Rocq establish both the internal consistency of our system and its equivalence to the LRM specification.

Based on this formal framework, we have developed some proposed text [6] that clarifies how expression bit-widths are determined, while preserving backward compatibility. We intend to submit our proposal for consideration by the IEEE SystemVerilog Working Group when it starts up again.

We hope that our system will be integrated into existing formally verified Verilog-consuming tools, such as the Vera verified equivalence checker [17] and the Lustig verified synthesis tool [12]. Both Vera and Lustig currently assume that their inputs are already 'elaborated', i.e. all intermediate expressions are explicitly annotated with their bitwidths. By enabling those tools to drop that assumption, our work could enhance the formal guarantees they provide.

Looking forward, we intend to expand the coverage and flexibility of this framework. While the current system requires concrete variable widths, we aim to introduce *parametric typing* to handle symbolic widths (e.g., determining a result width as $\max(4, N)$ with a size parameter N). Additionally, we plan to extend support to the full spectrum of SystemVerilog types, specifically incorporating floating-point and integer variables, as well as handling signedness.

Finally, we are exploring an alternative typing system that formalizes the notion of "context" informally introduced in section 2. A system in which AST nodes are first mapped to contexts, and then contexts are subsequently assigned bit-widths, may offer a more generic and intuitive formulation. This explicit definition of context could better align the formal semantics with the conceptual model used by hardware engineers.

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

Acknowledgements We're grateful to George A. Constantinides and Alastair F. Donaldson for their support. We also acknowledge financial support from the UK EPSRC via a PhD scholarship.

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. Proc. ACM Program. Lang. **9**(OOPSLA2) (Oct 2025). https://doi.org/10.1145/3763084, https://doi.org/10.1145/3763084
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. Desfrene, G., Corradi, Q., Pardalos, M., Wickerson, J.: Rocq formalization of systemverilog expression bit-width determination (May 2026). https://doi.org/10.5281/zenodo.19684093
7. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5) (May 2021). https://doi.org/10.1145/3450952
8. 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
9. 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
10. 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
11. IEEE: Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2023 (Feb 2024). https://doi.org/10.1109/IEEESTD.2024.10458102
12. 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
13. Lööw, A.: The simulation semantics of synthesisable Verilog. Proc. ACM Program. Lang. **9**(OOPSLA1) (Apr 2025). https://doi.org/10.1145/3720484
14. Meredith, P., Katelman, M., Meseguer, J., Rosu, G.: A formal executable semantics of verilog. In: Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). pp. 179–188 (2010). https://doi.org/10.1109/MEMCOD.2010.5558634

15. nebuchadnezzar_II: Lint tool is throwing an error about bit width when adding two 10-bit unsigned numbers and assigning to a 11-bit net. Electrical Engineering Stack Exchange (May 2023), https://electronics.stackexchange.com/q/665616
16. 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
17. Pardalos, M., Pozzi, L., Wickerson, J.: Towards mechanized verification of verilog equivalence checking. In: Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) (2025), https://capra.cs.cornell.edu/latte25/paper/9.pdf
18. 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
19. r/Verilog: Tutorial is wrong about truncation rules? (May 2026), https://www.reddit.com/r/Verilog/comments/1t0nb27/tutorial_is_wrong_about_truncation_rules/
20. The Rocq Development Team: The Rocq Prover (2025), https://rocq-prover.org/, version 9.0.0
21. Williams, S.: Icarus Verilog quirks: Unsized expressions as arguments to concatenation (2024), https://steveicarus.github.io/iverilog/usage/icarus_verilog_quirks.html#unsized-expressions-as-arguments-to-concatenation